



# **RITA+, AN SGML BASED DOCUMENT PROCESSING SYSTEM**

**A DISSERTATION  
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,  
FACULTY OF SCIENCE  
AT THE UNIVERSITY OF CAPE TOWN  
IN FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
MASTER OF SCIENCE**

**By  
Guido Zsilavec**

**Supervised by  
Associate Professor G. de V. Smit**

**Cape Town, March 1993**



The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

# Abstract

Rita+ is a structured syntax directed document processing system, which allows users to edit documents interactively, and display these documents in a manner determined by the user.

The system is SGML (Standard Generalized Markup Language) based in that it reads and saves files as SGML marked up documents, and uses SGML document type definitions as templates for document creation. The display or layout of the document is determined by the Rita Semantic Definition Language (RSDL). With RSDL it is possible to assign semantic actions quickly to an SGML file. Semantic definitions also allows users to export documents to serve as input for powerful batch formatters. Each semantic definition file is associated with a specific document type definition.

The Rita+ Editor uses the SGML document type definition to allow the user to create structurally correct documents, and allows the user to create these in an almost arbitrary manner. The Editor displays the user document together with the associated document structure in a different window. Documents are created by selecting document elements displayed in a dynamic menu. This menu changes according to the current position in the document structure. As it is possible to have documents which are incomplete in the sense that required document elements are missing, Rita+ will indicate which document elements are required to complete the document with a minimum number of insertions, by highlighting the required elements in the dynamic menu.

The Rita+ system is build on top of an existing system, to which SGML and RSDL support, as well as incomplete document support and menu marking, has been added.

# Acknowledgements

I would like to thank

The Department of Computer Science and its staff members, and especially Riël who introduced me to this fascinating topic which incorporates most of what I like about computer science,

My family for their support in all ways,

My friends and fellow students for those interesting discussions about life, the universe and everything,

And finally the Ocean and all the life in it for providing me with that ever so important diversion.

# Contents

<b>Abstract</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Batch formatters and markup . . . . .	2
1.2 Interactive systems . . . . .	4
1.3 The Rita system . . . . .	5
1.4 The Rita+ system . . . . .	6
<b>2 Rita System Overview</b>	<b>8</b>
2.1 Editor Features . . . . .	9
2.2 The Class Description Language . . . . .	11
2.3 Menu Calculation and Document Creation . . . . .	17
2.4 The introduction of SGML . . . . .	18
2.4.1 Why SGML? . . . . .	18
2.4.2 Why RSDL? . . . . .	20
<b>3 The Rita Semantic Definition Language</b>	<b>21</b>
3.1 Characteristics . . . . .	22
3.2 <i>If</i> statements . . . . .	24

3.3	Labels . . . . .	25
3.4	Named environment definitions . . . . .	26
3.5	Tag style definitions . . . . .	26
3.6	Example of using RSDL and SGML . . . . .	28
3.7	Calculating environments . . . . .	28
<b>4</b>	<b>Rita+ System Overview</b>	<b>32</b>
4.1	Document creation and incomplete documents . . . . .	32
4.2	Menu creation in Rita+ . . . . .	34
4.2.1	Menu marking . . . . .	35
4.2.2	Exceptions . . . . .	36
4.3	Using Rita+ . . . . .	36
4.4	Using the Rita Semantic Definition Language . . . . .	40
<b>5</b>	<b>Changing Rita into Rita+, a critical analysis.</b>	<b>42</b>
5.1	Why the change to SGML and RSDL? . . . . .	42
5.1.1	Problems with CDL . . . . .	44
5.1.2	Changing semantic languages . . . . .	45
5.1.3	Example of using RSDL and CDL . . . . .	46
5.1.4	Using the RSDL language . . . . .	50
5.2	Changing Rita into Rita+ . . . . .	50
5.3	The Rita+ system: implementation features . . . . .	51
5.4	The Rita+ system: performance considerations . . . . .	51
5.5	Arguments for a rewrite of Rita . . . . .	52
<b>6</b>	<b>Conclusion</b>	<b>54</b>
6.1	Further work . . . . .	55

<b>A</b>	<b>Standard Generalized Markup Language (SGML)</b>	<b>57</b>
A.1	Marking up documents . . . . .	57
A.2	Document Type Definitions . . . . .	58
A.2.1	Comments . . . . .	59
A.2.2	Defining document structure . . . . .	59
A.2.3	Entities . . . . .	60
A.2.4	Attributes . . . . .	60
<b>B</b>	<b>Default style definitions</b>	<b>62</b>
<b>C</b>	<b>Sample DTD's and semantic files</b>	<b>64</b>
<b>D</b>	<b>The Rita Semantic Language Document Type Definition</b>	<b>67</b>
<b>E</b>	<b>Rita Semantic Definition File for the Rita Semantic Language</b>	<b>71</b>
	<b>Bibliography</b>	<b>81</b>

# List of Figures

1.1	Structure for a book . . . . .	3
2.1	The Rita user interface . . . . .	10
2.2	Selecting <b>Front matter</b> . . . . .	11
2.3	Selecting <b>Title page</b> and <b>Title</b> . . . . .	12
2.4	Adding the <b>Title</b> . . . . .	13
2.5	Element definition in the Class Description Language . . . . .	14
2.6	Structure of the <b>if</b> statement . . . . .	16
2.7	DFA for some expression <b>S</b> . . . . .	17
3.1	Conditional expressions in environment declarations . . . . .	25
3.2	Definition of <i>condition</i> . . . . .	25
3.3	Named environment declarations . . . . .	26
3.4	Template for the tagstyle definition. . . . .	27
3.5	Template for the <i>for</i> -clause . . . . .	28
3.6	Tag styles for a simple poem . . . . .	29
3.7	SGML document type definition for a simple poem . . . . .	29
3.8	A poem marked up in SGML. . . . .	30
3.9	Poem displayed using semantic definitions . . . . .	30
4.1	Rita+ System Overview . . . . .	33



4.2	$\epsilon$ -DFA created from the DFA . . . . .	34
4.3	Marked menus . . . . .	36
4.4	Starting up Rita+ . . . . .	37
4.5	After inserting a key and a title . . . . .	38
4.6	Adding an author . . . . .	39
4.7	Adding a note . . . . .	39
4.8	Using Afrikaans semantic definitions . . . . .	40
A.1	A small SGML marked-up document . . . . .	58
A.2	A small SGML DTD . . . . .	59
B.1	Default root base environment for PC. . . . .	62
B.2	Default environment for PC. . . . .	63
C.1	Sample SGML DTD. . . . .	65
C.2	Sample RSDL semantic definitions . . . . .	65
C.3	Sample RSDL semantic definitions in SGML format . . . . .	66

# List of Tables

<b>2.1</b>	<b>Regular expression operators . . . . .</b>	<b>12</b>
<b>2.2</b>	<b>CDL Marking Instructions defined for Rita . . . . .</b>	<b>15</b>
<b>2.3</b>	<b>CDL Formatting Instructions . . . . .</b>	<b>15</b>
<b>4.1</b>	<b>BIBTEX fields for an article entry. . . . .</b>	<b>35</b>

# Chapter 1

## Introduction

In this thesis an SGML based document manipulation system is introduced. This system, called Rita+, provides the user with an interactive syntax directed way of creating and editing SGML based documents. The system furthermore allows the user to change the layout of the document on the screen by adding semantic actions using the Rita Semantic Definition Language, or to save files in a format suitable for input to batch formatters.

The Rita+ system is based on an existing system, Rita, created by the Computer Systems Group at the University of Waterloo. This system is in turn based on the work by Smit ([Smi87]).

This thesis investigates the issues involved in modifying the existing Rita document editing system (which was conceived as a front end to batch formatters) to an SGML based document processing system. The issues involved can be divided into two main areas, namely handling document semantics (layout), and adding support for incomplete documents. In detail these issues are:

- The suitability of using SGML to describe documents, and the effect of using SGML on the Rita system.
- The suitability of the Rita Semantic Definition Language to specify semantic actions quickly.
- The advantage of using SGML and the Rita Semantic Definition Language (RSDL) over the original Rita Class Description Language.

- The implementation of RSDL features in the Rita+ system.
- The implementation of incomplete document handling, menu calculation and marking.

The rest of this chapter introduces formatters and the concept of markup, batch and interactive systems, and furthermore shortly explains the ideas behind the original Rita system and the new Rita+ system.

The next chapter describes the evolution of the original Rita system, from the system designed by [Smi87], to the implemented Rita system, and the first prototype designed to process SGML documents. The reason for the change to SGML is also explained, as well as the subsequent need to introduce a new semantic language.

Chapter 3 explains the features of the Rita Semantic Definition Language.

In Chapter 4 the complete Rita+ system is introduced. Important issues such as incomplete documents, menu calculation and marking are described. Sample screens of the Editor are shown to highlight the use of RSDL as a semantic language.

Chapter 5 compares the different languages and explains why the existing semantic language was replaced with RSDL, and furthermore explains some of the issues involved in creating Rita+ and why it would be better to create a completely new system, rather than, as done, modify the existing one.

## 1.1 Batch formatters and markup

Batch formatters take as input a *marked-up* document, process it and produce high quality output (for an excellent paper on document formatters, see [FSS82]). Marking up a document is the process of including commands or *tags* in the user document which are recognized and understood by the formatter. In early systems, such as RUNOFF, tags were single whole lines starting with a period. Later systems such as T<sub>E</sub>X [Knu84] relaxed the placement of tags, and tags can now appear almost anywhere in the document. Systems such as RUNOFF and T<sub>E</sub>X are termed to be *procedural* formatters, as the user has to specify how each section of the document should be formatted. For example, it is up to the user to specify spacing such as headers and footers, margins and distance

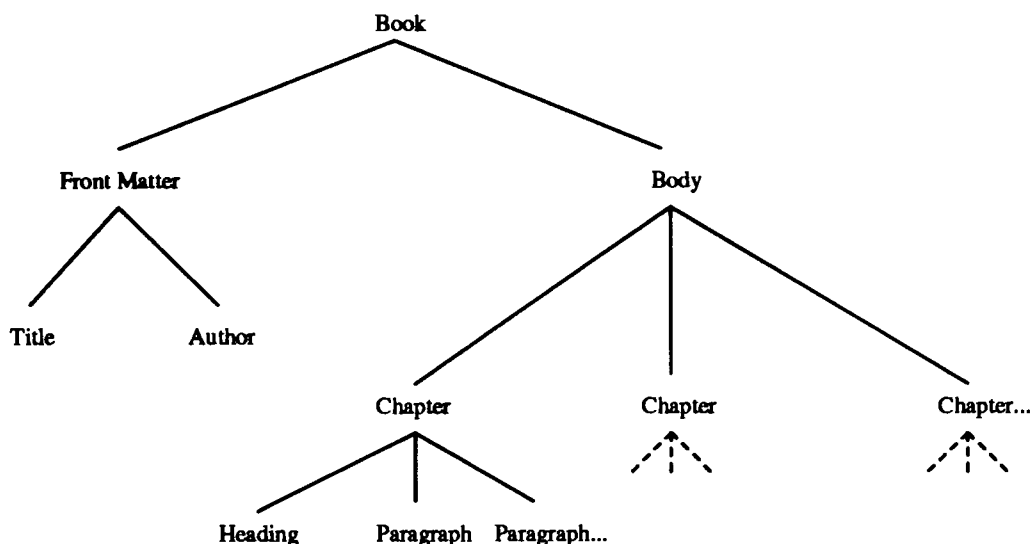


Figure 1.1: Structure for a book

between lines as well as font name, style and size. This gives the user freedom and power, but the learning curve is steep. It can be a formidable task for the new user starting with a formatter such as  $\text{T}_{\text{E}}\text{X}$ , and even the experienced user usually has to rely on a reference manual for information on infrequently used features.

As systems such as  $\text{T}_{\text{E}}\text{X}$  and  $\text{TROFF}$  [Sun90] are not easy to use, macro packages were introduced to simplify document creation and to help produce more uniform looking documents. Examples of such macro packages are the  $\text{\LaTeX}$  [Lam86] macros for  $\text{T}_{\text{E}}\text{X}$  and the  $\text{-ms}$  macros for  $\text{TROFF}$  [Sun90]. The tags remain, but now the user does not need to specify *how* the document should be formatted, but rather specifies *what* each section of the document is. For example, the user now indicates which sections are the headings, paragraphs, etc. and the formatter decides how to format each section.  $\text{\LaTeX}$  conveniently defines a few standard document styles, such as *article* or *report*, each providing a different layout and formatting style for the document. It is possible to modify these or add new styles if the user requires it. Formatters such as  $\text{\LaTeX}$ , called *functional* or *declarative* formatters, use the inherent underlying structure of most documents. For example, the structure of a book might consist out of front matter and a body. The front matter consists of a title and the authors name, while the body consists of chapters which in turn consist of a heading and a set of paragraphs. This structure is shown in Figure 1.1.

Declarative formatting languages have the advantage in that the user can now concentrate on the content of the document, and does not have to be concerned with the layout, which is

taken care of by the formatter [CRD87]. Furthermore, if documents are created according to a set layout standard, if this standard changes the user does not need to modify the document. In contrast, if the user were using a procedural formatter all the documents may need modification.

However even with declarative macro packages, batch formatters are still not easy to use, especially for the casual user. The document creation cycle easily becomes a tedious process, as the user has to run the document through the formatter and view the results before being able to modify the document and correct errors. Although the results of the formatting stage could be previewed on the computer itself, the user cannot modify the document in the display, and has to correct the original document, which can become cumbersome as tags are interspersed with the document and it becomes difficult to read and manipulate the document.

## 1.2 Interactive systems

Interactive systems removed the edit-format-preview cycle by displaying an approximation of the final result while the user is creating and editing the document. Especially later systems such as MS-Word, using graphical displays, variable font sizes, styles etc. provide the user with a WYSIWYG (what-you-see-is-what-you-get) display of the document. However, even though such systems are ideal for the casual user not requiring very high quality output, these systems are, compared to batch formatters, limited in their scope and abilities. A system such as TROFF has extensive macro facilities and over time specialized tools such as `pic` [Ker82] for typesetting pictures and `eqn` [KC75] for mathematical equations evolved. It is also possible to add macros to macros, such as the `grap` macros for typesetting graphs [KB86], which are based on the `pic` macros. With interactive systems, unless special tools are provided, it is not possible for the user to create these and add them to the system [Ker90]. For example, the MS-Word word processor provides a specialized equation editor, but only a simple graph editor. For more complex figures the user has to resort to using another specialized tool with possibly a completely different user interface, and import the resulting graph into the word processor, if that is at all possible. The advantage of ease of use is thus lost somewhat if the user has to learn to use another system, whereas macros for batch formatters are usually in a format

similar to the standard features of the formatter.

Another disadvantage of interactive systems is that it is often not possible to create macros for repetitive actions, or to even delegate these actions to a batch system. Furthermore, formatting is done continuously. Each insertion or deletion changes the appearance of the document, and the layout has to be recalculated. For example, line and page breaks change continuously while documents are being typed in. Batch formatters on the other hand process the complete document in one go. In systems such as  $\text{\TeX}$  line and page breaking occurs according to rules set up for typesetting.  $\text{\TeX}$  knows how to hyphenate words, and if it needs to break a word it tries to do so by minimizing the *badness* of the break. In interactive systems such as MS-Word this cannot be done, and it is possible that lines with large inter-word gaps are created.

More sophisticated interactive systems, such as Desk Top publishing systems give the user more freedom in laying out documents [Oma87]. Such systems are however often ineffective wordprocessors, forcing the users to import text from a word processor. Systems such as these are not aimed at the casual user, but more at those already involved in publishing and who understand the rules for document layout and presentation.

The conclusion is that both systems have their advantages and disadvantages. The solution would be to create a system which incorporates the ease of use of an interactive system with the power of a batch formatter [CS86].

### 1.3 The Rita system

The Rita system is such a solution [PM89]. The Rita system combines an interactive editing environment with the ability to export marked-up documents which can be processed by batch formatters to produce high quality output [CMS91].

Rita is a syntax directed editor. This means that documents are created according to a predefined structure. Syntax directed editors are often used for writing computer programs, as programming languages have a set syntax or structure. Documents also show structure as explained previously, and as can be seen in Figure 1.1, documents do not only have structure, but the elements of the structure are also in an hierarchical order. Thus, while chapters contain paragraphs, paragraphs can never contain chapters. This feature of

documents has been used by several editing systems [Qui89], and with these systems the user can create documents according to the underlying structure. When using Rita, the user defines the document structure, and can then create documents according to it. Rita can furthermore display documents in a what-you-see-is-*almost*-what-you-get approximation, by allowing the user to define *semantic actions* for each element in the document structure. These semantic actions determine the look and layout of the document on the screen, and add tags to the document when exporting the document to a batch formatter.

The language which Rita uses to describe the structure and the semantic actions was defined for the system described by [Smi87]. The Rita system, as implemented by the Computer Systems Group of the University of Waterloo, used a subset of the language defined, but added a few features of its own.

## 1.4 The Rita+ system

The Rita+ system is based on Rita, but the aim of the system changed. Rita+ has become an SGML document manipulation system, in contrast to Rita which is a front end for batch formatters.

SGML is defined as “a language to define the input representation of documents, nothing more and nothing less.”<sup>1</sup>. The language itself is fully declarative. The structure and its elements are defined in a document type definition (DTD). The element names form the tags which are used to mark up the document. Apart from the structure, SGML does not interpret any part of the document, and thus does not know how to display or represent it. A typical SGML parser takes a user document and a document type definition, parses the document according to the type definition and either accepts the document as being correct, that is, conforms to the structure definition, or rejects it. An overview of the SGML language, markup and document type definitions, is given in Appendix A.

SGML, or Standard Generalized Markup Language, was designed as a document markup and interchange language. Given the document structure definition and the user document, any SGML user can understand the structure of the document. However, SGML

---

<sup>1</sup>Sam Wilmott of Exoterica in conversation with John McFadden, December 1989. From *comp.lang.sgml* on UseNet



suffers from some of the same problems batch formatters experience, namely, user text is interspersed with tags, making the marked up text difficult to create, read and modify. Furthermore, when creating an SGML document the user has to know the underlying document structure and thus understand SGML. Creating an SGML document becomes similar to creating a T<sub>E</sub>X document: edit and run through the SGML parser, and depending on the results, correct the errors. Rita+ helps the user to manipulate SGML documents by separating the tags from the text, that is, separating content from structure. Rita+ allows the user to create documents according to the structure, without having to know the details of the underlying structure or possibly even SGML. As SGML does not format text, Rita+ furthermore allows the user to specify semantic actions for each element in the document structure. These semantic actions are aimed primarily at the display of the document on the computer screen, but it is possible, as in the original Rita system, to add tags to the document as part of the semantic actions, and export the document so it can serve as input for a batch formatter.

The Rita+ system differs from the original Rita system in several ways. Mainly the aim of the system has changed, from being a batch formatter front end to being an SGML document manipulator. This change has influenced the whole system. For example, the Rita system uses its own document structure and semantic action description language, whereas the Rita+ system uses SGML to describe structure, and uses a new, more powerful language, to describe the semantic actions.

The Rita+ system also incorporates ideas present in [Smi87] but not implemented in the Rita system, as well as certain other features needed to support SGML.

## Chapter 2

# Rita System Overview

The Rita system was designed by [Smi87] as a front end for batch formatters, offering the user syntax directed creation of documents, coupled with document semantics which control the display of the document and the ability to export documents marked-up in a manner suitable for batch formatters.

The Computer Systems Group at the University of Waterloo took the ideas of [Smi87] and created the Rita system. The basic ideas of the Rita system remained, but a few changes were made. The system does not implement some of the ideas proposed in [Smi87], such as sub-sequence incomplete document support and exceptions to regular expressions, but adds GML support. GML (Generalized Markup Language) is the formatting language used by IBM for its Document Composition Facility (DCF). It is also used by WATCOM for its Information Workbench range of products, of which Rita is a member, which are created by the Computer Systems Group at the University of Waterloo [PM89]. With Rita the structure and semantic actions are defined using the Class Description Language, and are saved in a file called a *class description*. Each class description describes the structure and semantics of a set of documents. The Rita system consists of two parts. The Rita Class Generator (RitaCG) processes a class description and converts it to a more compact representation. The Editor loads this file, the intermediate file, to allow users to create documents according to the structure defined within it.

## 2.1 Editor Features

Figure 2.1 illustrates the user interface of a typical screen as seen by the user during the editing process is shown. The screen can be divided into four main components:

1. The *main menu* is used for static menu functions, such as loading and saving files, spell checking etc. In Figure 2.1 the menu for *cursor* is selected, which allows the user to select the size of the cursor when it is in the edit window. This feature allows the user to select large sections of the document quickly. The *extend* feature allows the user to drag the cursor to select text, both in the edit and the structure window. The edit mode is by default *insert*, but can also be *overwrite*.
2. The *dynamic menu* shows those elements which can be inserted at the current cursor position. This menu changes according to the cursor position within the document structure, and according to the class description currently active.

For each menu entry both the element name and its shortcut key are shown. To select an item the cursor has to be in the structure window, and the user can either press the shortcut key, or select a menu item with the mouse cursor. It is possible that several menu items have the same shortcut key. If the user selects one of these items using the shortcut key, a sub-menu is created listing only those menu items with the same shortcut key. The user can then select the required item by specifying the *number* corresponding to the required menu item.

3. The *structure window* shows the structure of the document currently being created, by displaying the relevant start and end tags corresponding to the document structure element. The indentation shows the document hierarchy.
4. The *edit window* is where the user types in and manipulates the content of the document. It is possible to shift the vertical bar dividing the structure and edit windows, reducing the size of one window and increasing the other.

A few minor components are also visible:

- The square brackets show the text currently in the delete buffer. The contents of this buffer can be re-inserted by the user. In Figure 2.1 the buffer is empty.

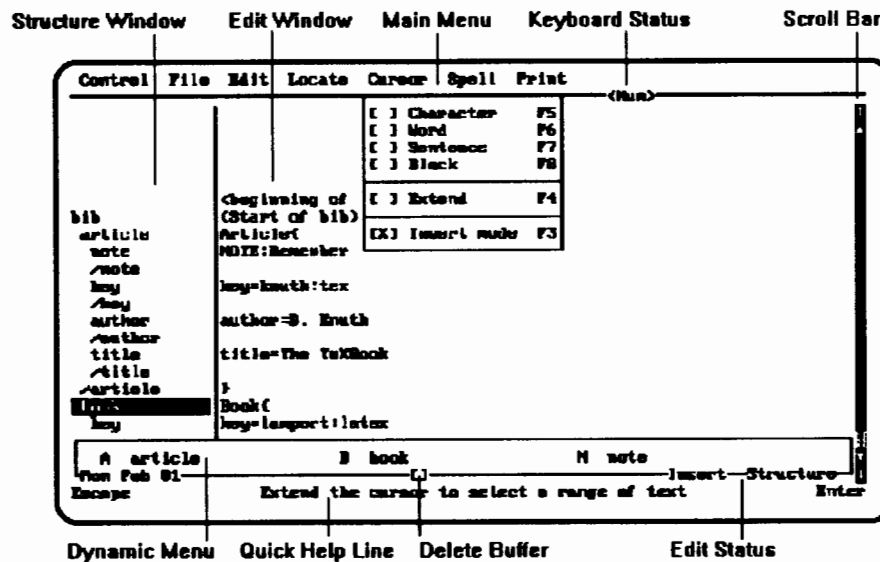


Figure 2.1: The Rita user interface

- The quick help line gives a short description of the currently allowable action. In Figure 2.1 the menu for *cursor* is currently active, and the quick help line describes the *extend* feature.
- The *edit status* shows whether the current mode is *insert* or *overwrite*, and whether editing is being done in the structure or edit window.
- The *keyboard status* indicators show which of *caps-lock*, *num-lock* or *scroll-lock* are set, in this case only *num-lock* is set.
- The scroll bar can be used to browse quickly through the document. It is possible to browse line by line, or page by page, or by using the slider manually.

To create documents the user selects a document element from the dynamic menu. In Figure 2.2 the dynamic menu contains the document elements **Body** and **Front matter**. These are the elements which can be inserted before the current cursor position. As can be seen in Figure 2.2 the cursor is on the document element **oGDOC** in the structure window. In Figure 2.3 the element **Front Matter** has been selected, and the display shows the menu with the elements that can be inserted. From the menu in Figure 2.3 the element **Title page** is selected, and from the subsequent menu the element **Title**. In Figure 2.4 the result of this selection is shown, and as **Title** is a *text level* element, the user can now

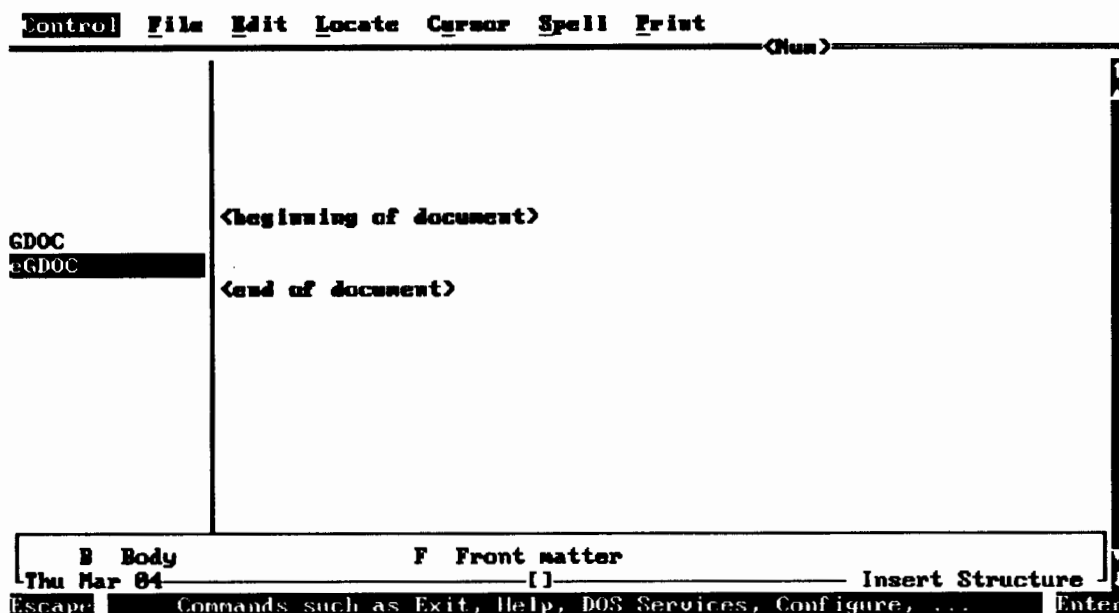


Figure 2.2: Selecting Front matter

type in text in the editing window, as shown.

## 2.2 The Class Description Language

The Class Description Language (CDL) used by Rita [Pia89] is based on the language defined by [Smi87], but only a subset of the language was implemented. The Rita system uses GML to save and load files, and this was reflected in some aspects of the implemented language.

A Class Description file consists of a collection of document element definitions. Each document element definition consists of a structure definition section, a semantic action section, menu and structure tag section and a GML unparsing section.

Structure is defined using regular expressions. The regular expression operators are shown in Table 2.1.

For example, the definition for the structure of a book as shown in Figure 1.1 can be expressed using regular expressions as follows:

```
Book      = FrontMatter Body
FrontMatter = Title Author
Body      = Chapter*
Chapter   = Heading Paragraph*
```

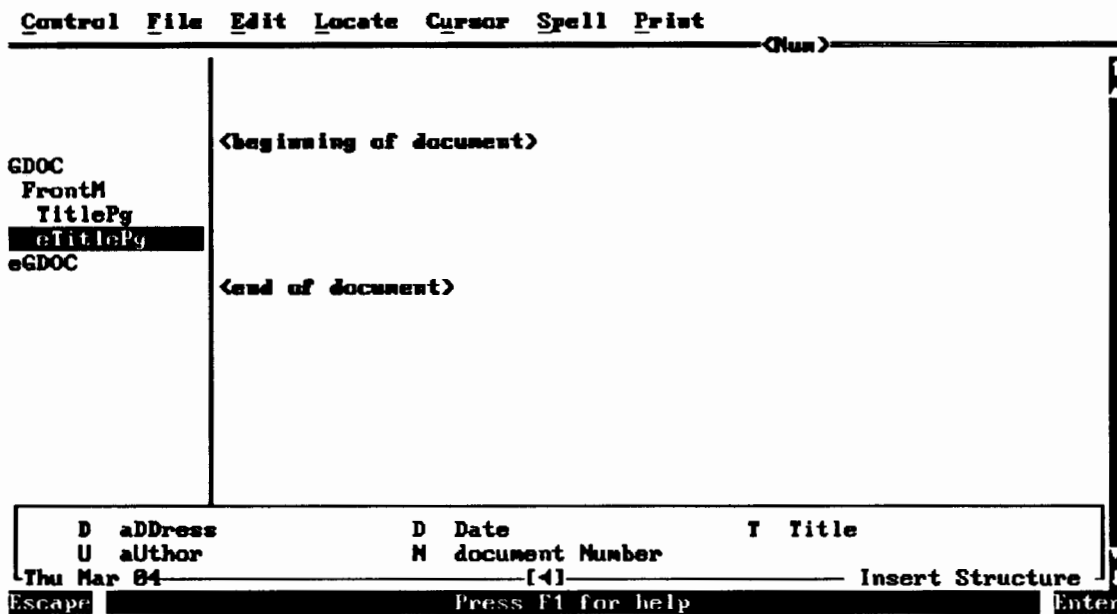


Figure 2.3: Selecting Title page and Title

<b>sequence:</b>	All elements must occur, in order.	$A B$
<b>or:</b>	Only one of the elements must occur.	$A B$
<b>and:</b>	All elements must occur, but in any order.	$A\#B$
<b>optional:</b>	Element occurs optionally.	$A?$
<b>repetition:</b>	Element is repeated zero or more times.	$A^*$
<b>plus:</b>	Element is repeated one or more times.	$A^+$

Table 2.1: Regular expression operators

In the system by [Smi87] it was possible to specify *exceptions* to a regular expression. Exceptions are *inclusions* and *exclusions*, where an element can either be included within a regular expression, or excluded from a regular expression. For example, one could define a **footnote** element and add it to the definition of a book. Footnotes can appear almost anywhere, except within footnotes themselves. The easiest way to specify this is by making the footnote an *inclusion* at the top level element, **Book** in this case, and to make it an *exclusion* in the footnote element. Inclusions are denoted by adding a plus before the element name, and exclusions by adding a minus. The structure definition of a book could thus be modified as follows:

```

Book      = FrontMatter Body      +footnote
FrontMatter = Title Author
Body      = Chapter*
Chapter   = Heading Paragraph*
Footnote  = Text                  -footnote

```

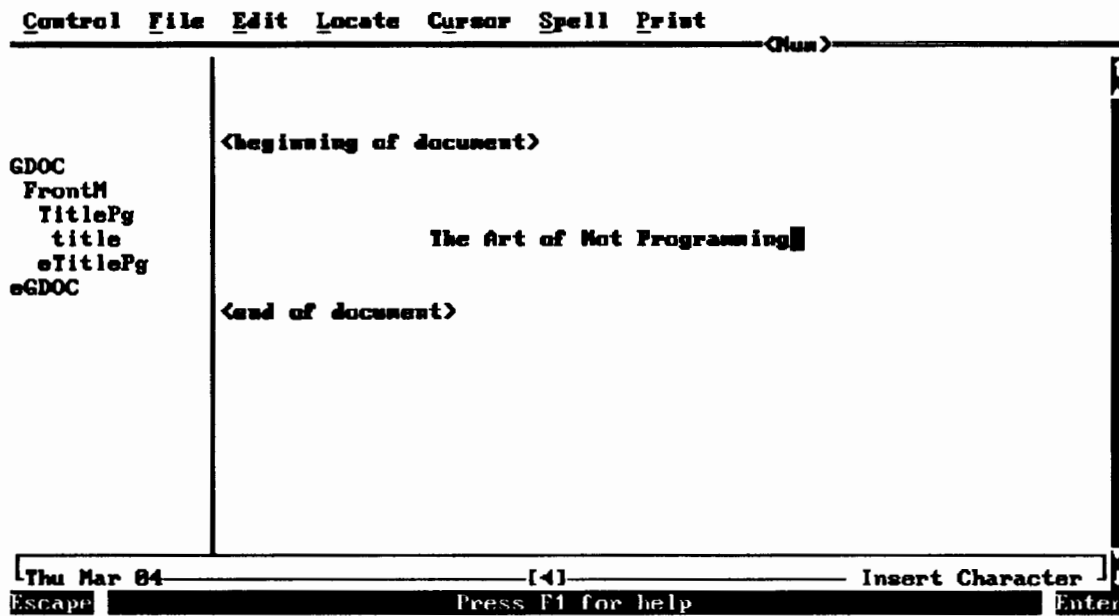


Figure 2.4: Adding the Title

In Figure 2.5 the element definition for *Book* is shown. The line numbers are added to clarify the different sections, and are not part of the language.

In line 1 the element name and structure using regular expressions are defined. As shown before, the definition of *Book* is *FrontMatter* followed by *Body*. Lines 2 to 13 inclusive defines the semantic action section which in the Rita CDL is divided into *schemes*, each of which is divided into a number of *methods*. In this example two schemes are defined, namely *display* and *gml*. The *display* scheme is always required as it defines the semantic actions required to represent the document on the display. As Rita saves files using the GML markup language a *gml* scheme is also required, as it specifies the tags to be used when saving a user file. Within a scheme different *methods* may be defined, each of which can be used to perform different semantic actions. In Figure 2.5 only the default *Standard* method is used for both schemes. Although the braces after *Standard* suggest variables and parameter lists, these are not supported in the Rita CDL.

Lines 4 and 5, as well as 10 and 11, define the actual semantic actions, or *unparsing programs*, as they are called in CDL. Each unparsing program is a sequence of commands, which are executed on a transition to the next element. For example, from element *Book* the first transition would be to element *FrontMatter*, and the second one to element *Body*. With CDL it is possible to specify the unparsing program which should be executed

```

1  Book = FrontMatter Body
2    [ /display/
3      Method Standard()
4        initial : @nl "Start of Book",
5        final   : @nl "End of Book"
6      endMethod
7
8    /gml/
9      Method Standard()
10       initial : @nl ":book.",
11       final   : @nl ":ebook."
12     endMethod
13   ]
14   <"book", "endBook", "B", "Book">
15   @ "book", @req, @noattr, @req, @nofmt @
16   ;

```

Figure 2.5: Element definition in the Class Description Language

before any transition (*initial*), after all transitions (*final*) or before a specific transition (not shown).

For example, as shown on line 4, before the transition to element *FrontMatter* a new line is started, and “Start of Book” is displayed on the screen, while line 5 shows that, after the transition to element *Body*, on a new line, “End of Book” is displayed.

Rita separates unparsing instructions into *marking* instructions which do not generate output, but may change the formatting environment, and *formatting* instructions which generate output. Table 2.2 shows the marking instructions, while Table 2.3 shows the formatting instructions.

The `@method()` marking instruction is used to specify which method is to be used when formatting the content element on the transition to that element. If no explicit method is indicated the `@Standard()` method is used by default, as is shown in this example. By defining different methods in the definition on an element, it is possible to perform different semantic actions depending on the context of the transition to that element.

Variables are implemented in a crude manner. A single variable (or counter) is associated with each document element defined. The marking instruction `@incvar` increases the parent element variable on a transition from the current element to one of its children. The `@emitvar` formatting instructions outputs the value of the parent element variable as



<b>@method()</b>	Use the method named <i>method</i> when formatting.
<b>@bold</b>	Set the text to bold.
<b>@ebold</b>	Do not set the text to bold.
<b>@underline</b>	Underline the text.
<b>@eunderline</b>	Do not underline the text.
<b>@normal</b>	Set the text back to normal, ie. not bold or underline.
<b>@left</b>	Left justify the text. This is the default.
<b>@right</b>	Right justify the text.
<b>@center</b>	Center the text
<b>@ecenter</b>	Do not center the text.
<b>@lm=(<i>n</i>)</b>	Set the left margin to <i>n</i> characters.
<b>@lm+(<i>n</i>)</b>	Increase the left margin by <i>n</i> characters.
<b>@lm-(<i>n</i>)</b>	Decrease the left margin by <i>n</i> characters.
<b>@rm=(<i>n</i>)</b>	Set the right margin to <i>n</i> characters.
<b>@rm+(<i>n</i>)</b>	Increase the right margin by <i>n</i> characters.
<b>@rm-(<i>n</i>)</b>	Decrease the right margin by <i>n</i> characters.
<b>@fill_chars</b>	Do not word-wrap text on reaching the right margin
<b>@fill_words</b>	Perform word-wrap on the text on reaching the right margin.
<b>@hide</b>	Set the hide attribute for the current element.

Table 2.2: CDL Marking Instructions defined for Rita

<b>"string"</b>	Output the string <i>string</i> .
<b>@attr</b>	Output the element's attributes.
<b>@bar</b>	Output a horizontal bar. Only possible in the <i>display</i> scheme.
<b>@nl</b>	Output a new line.
<b>@vs(<i>n</i>)</b>	Output <i>n</i> vertical space lines.

Table 2.3: CDL Formatting Instructions

```

if <condition 1> then <unparsing program 1>
elseif <condition 2> then <unparsing program 2>
...
else <program n>

```

Figure 2.6: Structure of the `if` statement

a string.

Rita also defines a simple conditional *if* statement. The format of the *if* statement is shown in Figure 2.6. The conditions are `@empty` and `@firstsib`. The `@empty` condition is true if the current element has no content. The `@firstsib` condition is true if the current element is the first element within the context of its parent.

Line 14 defines the label definition. Label definitions are used to display the element name in the structure window, and to create the menu entries. The first two entries are the start and end tag for the structure display of the element. The third entry is the menu shortcut selection character, and the fourth the menu name for the element. It is possible to group elements. For example, if one had the elements *ordered-list*, *un-ordered-list*, *numbered-list* and *bullet-list*, these could be put into a group called *lists*. Thus instead of having four menu items only one is necessary, and when selecting element *lists* the other four elements are shown. This feature saves spaces by reducing the size of menus.

Furthermore it is possible to specify `@hide`, which causes the menu item to be hidden from the menu, unless the Editor is in *verbose* mode.

Line 15 shows the parsing definition. The format is as follows:

```

@ "scan_tag", start_tag_req, attr_allowed, end_tag_req, format_type @

```

Parsing definitions are used by Rita to read in a file. Files are saved according to the tags defined in the *gml* scheme. Each parsing definition defines the information necessary to read in the element. The scan tag defines the element in the document. Start and end tags may be omitted if this can be done un-ambiguously. This is done by specifying `@noreq` for `start_tag_req` and `end_tag_req`. If a tag is required `@req` is specified. Similarly attributes may or may not be allowed by specifying `@attr` or `@noattr` for `attr_allowed`. The `format_type` may be `@nofmt` if the element, as in this example, has no text, or it

may be **0blk** if it has a block of text associated, **0txt** if the element contains text and is a text level element, or it may be **0ln** if there is only a single line of text associated with the element.

## 2.3 Menu Calculation and Document Creation

The system designed by [Smi87] supported *sub-sequence* incomplete documents. This means that documents can be created in a non-sequential fashion. For example, using the regular expression  $S = b^* a (a \mid b)^* c (a \mid b)^*$ . A sub-sequence incomplete document would be **bbc**, as it misses required element **a** anywhere before element **c**. The implementation of Rita did not support sub-sequence incomplete documents but only a subset of it, namely *tail-sequence* incomplete documents. Documents had to be created in a front to back sequential fashion, with only elements at the end of each document section possibly missing. Thus, it would not be possible to create the document **bbc**, but it would be possible to create **ba**, with the element **c** missing at the end.

The Class Generator converts the regular expressions defining the document structure into corresponding deterministic finite state automata (DFA's). A finite state automaton for the regular expression  $S$  is shown in Figure 2.7.

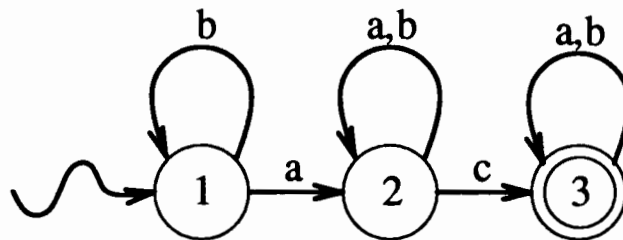


Figure 2.7: DFA for some expression  $S$

The menu calculation routine uses the finite state automata representation to calculate menus. For example, if elements are inserted after a state, in state 1 the menu consists of the elements **a** and **b**, namely the elements corresponding to the transitions out of state 1. Similarly in state 2 the menu consists of elements **a**, **b** and **c**. As menus are calculated according to the current state within the automata it can be seen that it is not possible to create the sequence **bbc**, as there is no transition on element **c** from state 1.

## 2.4 The introduction of SGML

The Computer Systems Group decided to change the existing Rita system and to convert it into an SGML document processing system. The changes involved in doing this were the re-writing of the Class Generator, which now instead of accepting a class description accepts an SGML document type definition as input. The Editor only needed a few minor changes for it to save and load SGML files. However, as SGML does not provide document semantics, the Editor's crude built-in semantic actions were all that was available for displaying documents. It was then decided that instead of modifying the semantic action section of CDL a new language, RSDL, was to be used as there were some deficiencies in CDL.

### 2.4.1 Why SGML?

SGML is an international standard for document markup and interchange. Since 1986 when the standard was released [ISO86], SGML has grown in acceptance and in the number of applications. One of the largest users of SGML is the U.S. Department of Defense in their CALS (Computer-aided Acquisition and Logistic Support) programme [Bar89]. This programme was created to control the paperwork associated with the design, manufacturing and maintenance of weapon systems. The advantage in this case is that the manuals created are both human and computer readable. Thus, it can be read as a normal manual, yet it is possible to perform queries and searches on the text, as with a database. As the marked-up text can be used by both humans and computers it is not necessary to duplicate the information, avoiding the problems usually associated with duplicated data.

Other SGML applications are the encoding of the Oxford English Dictionary and the McGraw-Hill Encyclopedia of Science and Technology [Bar89]. Publishers are also starting to use SGML to mark-up books, making it easier to publish books in different formats [Hay92].

With the acceptance of SGML standard document type definitions are also appearing, such as those by the Text Encoding Initiative [CMB90] and those by the American Association of Publishers [Ass88c]. The increasing acceptance of SGML can also be shown with the appearance of articles on SGML in popular magazines such as BYTE [Wri92]

and UnixWorld [Hay92].

One of the problems with SGML is the lack of tools for creating and manipulating SGML documents. A few tools exist, such as The Publisher and SGML-EDITOR by ArborText [Gro91], as well as *DynaText* by Electronic Book Technologies [DeR90]. *DynaText* is a system for the online delivery of books. *DynaText* gives publishers the freedom to view documents in a variety of formats, and to link sections of the document with hypertext links. For example, a footnote can be retrieved quickly by accessing its reference [RD92]. SGML editors do exist, such as the commercial system introduced in [Miz91], but this editor shows tags and user text in one window, making it difficult to read text quickly, even though tags are distinctly marked as such. The display does show where elements can be inserted, but does not format the text in any way. It is possible to export documents as a  $\text{\LaTeX}$  file by providing a conversion file which describes the relation between the elements defined in the SGML document type definition and  $\text{\LaTeX}$  commands. The system described in [WvV91] is not an editor, but allows the user to add programs to the SGML document type definition (using the C programming language) which can be used to output for example  $\text{\LaTeX}$  commands for the user file, thus allowing the user to export an SGML marked-up file to  $\text{\LaTeX}$ . The disadvantage of this system is that the actual SGML DTD has to be modified or copied which results in the usual problems of data duplication and inconsistencies, and the system is not interactive, as it is fully batch driven.

The problem with most of the existing tools is that they are either limited in their abilities, or they are very specialized in nature. The ArborText products are very CALS oriented, and *DynaText* is aimed at the publishing market. The system described in [Miz91] works mainly in Japanese.

The new Rita+ system is designed as a smaller but more general tool, which can run on a variety of systems with limited resource requirements. The new system allows the user to manipulate SGML documents quickly without requiring too much previous knowledge of SGML or of the document type definition currently being used.

The structure section of the language defined by [Smi87] was based on the SGML structure definitions, and they are essentially the same. The structure section of the Class Description Language could thus be replaced easily with SGML.

### 2.4.2 Why RSDL?

With the introduction of SGML to describe documents the structure section of a class description became obsolete. Now the problem was whether to adopt the semantic section of CDL or to create a new language. Due to some deficiencies in CDL it was decided to create a new language.

The main problem with CDL is its verbosity, and the need to distribute semantic actions for one element definition over the definitions of various elements. This, together with the implementation of CDL and Class Generator made it hard to assign semantic actions quickly to a new document type definition, one of the aims of the new system.

The initial RSDL design [Smi92] was strongly influenced by the ideas for FOSI (Formatting Output Specification Instance), a Department of Defense standard conceived for the CALS project [USA88].

The following chapter explains the RSDL language in detail, and applications of the language using Rita+ are shown in Chapter 4 using sample screens to show how an RSDL semantic file affects the display of a document.

## Chapter 3

# The Rita Semantic Definition Language

The Rita Semantic Definition Language (RSDL) is used to assign semantic actions to a document. Semantic actions are associated with the elements defined in the corresponding SGML DTD.

When Rita+ processes an SGML document for formatting, a set of formatting characteristics such as font style, size, margins, etc., determine the layout of the current element. This set of formatting characteristics forms the *environment* in which the document is formatted, and it defines the visual appearance of the document.

A base environment is instantiated at the start of the document and remains in effect throughout the document. The semantic description associated with each element can modify the value of the characteristics of this environment, but the changes remain in effect only for that element and possibly its sub-elements. The changes to the characteristics can be either relative or absolute. If the change is relative, the value of the current environment is used and possibly modified; if the change is absolute, a new value is instantiated.

Rita+ defines a built-in absolute environment which serves as the base environment for the entire document. This environment is called the *root base* environment. The user can override this environment to a certain extent by providing a new root base environment, but can never eliminate it. For example, the root base environment is required if the user specifies a relative value in its root base. In this case the relative value is evaluated using

the default root base as a basis. This ensures that the base environment for the document is always an absolute environment. If all the values in the user's root base environment are absolute, then the default root base environment is overwritten completely. The default root base environment is defined in Figure B.1, in Appendix B.

Semantic actions are saved in a Rita Semantic Definition (or RSD) file. An RSD file consists of a set of *schemes*, each scheme describing one set of environment definitions. One scheme is required, namely the one for the display, which describes the formatting or layout of the document on the display screen. Each scheme may consist of *named environment definitions* and *tagstyle definitions*. A named environment definition can be used in other named environment definitions and in tagstyle definitions. Tagstyle definitions are environment definitions associated with specific elements. Within styles, not all characteristic fields of the environment have to be specified. Missing items are either inherited or will take on a default value.

A default environment is defined in Rita+. This environment is used for elements which do not have an explicit style, and if an element does not define some characteristics then the values defined in the default environment are used. The default environment is shown in Figure B.2, in Appendix B. As with the *root base* environment, it is possible for the user to define a different default environment. And as with the root base environment the built-in default environment is never eliminated completely, as it is required to ensure that each environment has a value in each of its characteristics fields.

### 3.1 Characteristics

Following are the characteristics which form a complete environment.

**Left indent** The left margin for the current element.

**Right indent** The right margin for the current element.

**First indent** The indent for the first line of the element.

**Line length** Total line length, including left, right and first indents.

**Line height** Distance between the baselines of two elements.



**Prespace** Total vertical space before the current element, in addition to line height.

**Postspace** Total vertical space after the current element, in addition to line height.

**Tabs** Tab settings, which can either be absolute (a set of tab settings) or relative to the left margin (first tab position and increment).

**Font name** Name of the current font. Architecture dependent.

**Font size** Size of the current font. Architecture dependent.

**Font style** Style of the current font. Can be *plain*, *bold*, *italics* or *underline*.

**Form** Either *block*, *inline* or *page*, used for the determining layout of document elements.

If the form is *block* it is possible to indicate the type of “edges” the block has, where an edge is the start or the end of a block. The type of edge determines whether the block starts or ends on a line. The type of edge may be:

**Smooth** Always start (end) a line.

**Sticky** Do not start (end) a line unless the previous (next) element edge is a smooth edge.

**Rough** Start or end a line unless the previous (next) element edge is a sticky edge.

**Justify** One of either *left*, *right*, *both* or *centered*.

**Translucent** If an element is translucent its environment cannot be inherited by its children, and it is thus “invisible” to the children.

**Suppress** The current element and its children are not displayed, unless the Editor is in *verbose* mode.

**Savetext** Define a construction rule for a string consisting out of text and variable names. The result is stored into a string variable, which can be displayed using *puttext*.

**Puttext** Output text and the values of variables. Output can be either *before* or *after* the element, or on both sides, and according to a specified environment which is only valid for that *puttext*.

**Enum Enumerate.** Define a variable, its initial value and its increment and a “within” element type which allows the counting of elements within other elements. Thus, for example, it is possible to enumerate each item within a list, or chapters within a book.

Numeric characteristics (left, right and first indent, line length and height, pre- and post-space and font size) can be either absolute or relative. If the value is relative it is preceded by a + (or -), in which case the parent’s value is inherited and increment (or decremented). A value of +0 inherits the parents value directly. Unless the value of the characteristic is set to 0 or +0, a unit is required. The units of numeric characteristics can be in *inch*, *centimeters*, *points*, *lines* or *spaces*, depending on the characteristic. Thus, for example, fonts may be only in points, centimeters or inch, line height only in inch, centimeters or lines. Values are converted automatically depending on the display capabilities of the machine running Rita+.

For non-numeric characteristics it is possible to either specify one of the given values, or to specify *inherit*, in which case the parent’s value is inherited.

As there are two ways of setting tabs, either a list of tab settings or a first tab and increment, the values within the tab have to be absolute and may not be relative.

## 3.2 If statements

The *if* conditional statement allows a style to set an environment according to its context within the document tree. Figure 3.1 shows the structure of an *if* statement, and Figure 3.2 shows the structure of the condition, using extended BNF notation.

In Figure 3.2 the **element** may be any element defined within the corresponding SGML document type definition. **IPRECED** refers to the immediately preceding sibling, while **IFOLLOW** refers to the immediately following sibling. As can be seen it is possible to check for the existence of any parent or sibling, by specifying for example if **IPRECED** is **NULL** to check if the current element is the first child in the context of its parent, or to check for the existence of a specific parent or sibling. It is furthermore possible to negate the check, and also to check for the existence of any ancestors and siblings at a specific location, by

```

if <condition> then
  <environment specification>
elseif <condition> then
  <environment specification>
  ...
else
  <environment specification>
elseif

```

Figure 3.1: Conditional expressions in environment declarations

```

condition := ("PARENT" | "IPRECED" | "IFOLLOW")+ "NOT"? (element | "NULL")

```

Figure 3.2: Definition of *condition*

specifying for example, **if IPRECED of PARENT of PARENT** to search for the immediate precedent of the grandparent.

Rita+ only defines the conditions **PARENT**, **IPRECED** and **IFOLLOW**, although many more options are possible, such as for example **PRECED** or **FOLLOW**, to refer to any preceding or succeeding siblings at any location, and **FIRSTCHILD**, to refer to the first child within an element. As it is a relatively trivial exercise, these and other possible additional conditions have not been implemented. In Rita+ documents are represented internally in their hierarchical tree structure, and any other conditional checks only require more extensive searches in the document tree.

### 3.3 Labels

Labels define the text for start and end tags used in the structure section of the display, as well as the menu label and its shortcut key. If **tagname** is specified, the tagstyle name is used. This name usually corresponds to the name of an element defined in the corresponding SGML DTD. In this case the start tag would be simply the tagstyle name, the end tag would be the tagstyle name preceded by a slash. The menu would again just be the tagstyle name and the shortcut its first letter.

```

Environment: bold-text
  Font name   : "cmr10"
  Font size   : 10 pt
  Font style   : bold

Environment: heading
  Use environment "bold-text"
  Justification : centered
  Form          : line
  Postspace     : 1 line

Environment: quote
  Font name   : "cmi8"
  Font size   : 8 pt
  Font style   : italics
  Form        : block ( smooth, smooth )
  Puttext     : before ""
               after  ""

```

Figure 3.3: Named environment declarations

### 3.4 Named environment definitions

Named environments, which are essentially macros, can be used within other named environment or within tag styles through the use of the `Use environment` command. Within style definitions it is possible to include all the characteristics, as well as constructs such as *if* statements and labels. In Figure 3.3 a few example named environment definitions are shown.

### 3.5 Tag style definitions

Tag styles are associated with individual elements. Figure 3.4 shows the template for a tagstyle. Both `when` and `otherwise` statements are optional, and it is possible to have only the optional environment specification. Each tagstyle consists of an optional environment specification and a set of *when* statements. A *when* statement is a conditional expression, similar to the *if* statement, which refers to the context of the element to which the tagstyle is associated.

When an element needs to be formatted, for example on element creation, the *when*

```

Style for: <list of element names>
<Optional environment specification A>
When <context expression>
    <environment specification B>
When <context expression>
    <environment specification C>
.
.
Otherwise:
    <environment specification N>
<labels>

```

Figure 3.4: Template for the tagstyle definition.

expressions are evaluated. The environment for the elements thus consists of the optional environment specification (environment specification *A* in Figure 3.4), and the environment specification corresponding to the first context expression which evaluates to true. If none of the expressions is satisfied the default *otherwise* environment specification (environment specification *N* in Figure 3.4) is used. If a characteristic is multiply defined, which is possible if it is defined for example within the optional environment specification and within a *when* statement, only the last definition is used. This applies for both absolute and relative values. For example specifying +5 and +4 causes the characteristic to take the value +4 and not the accumulated value of +9. Similarly, if the first value is absolute and the second value relative then the relative value is used.

Each environment specification in a tagstyle can consist out of characteristics, *if* statements, style definition references as well as *for*-clauses.

A *for*-clause allows the style of an element to define environments specific for a set of children. These environments definitions do not change the current environment, and are only valid for the children. This construct is the equivalent to specifying the corresponding *when*-clause in each child mentioned in the list, but it avoids unnecessary repetition if the same environment has to be set up for a number of children. Figure 3.5 shows the template for a *for*-clause.

```
For child <list of immediate descendents>
  <environment specification>
```

Figure 3.5: Template for the *for*-clause

## 3.6 Example of using RSDL and SGML

In Figure 3.6 the tagstyle definitions for a poem are shown, which correspond to the SGML document type definition shown in Figure 3.7. The styles for the elements **title**, **author** and **stanza** all use named environments defined in Figure 3.3. The *puttext* command in the tagstyle for the element **author** declares its own environment to ensure that the text “**Author:**” does not appear in bold type as does the actual author name. The **stanza** tag style uses *when* statements to determine the prespace and the first indent, depending on whether it is the first **stanza** or a subsequent one. The **line** style is very short, and only redefines the shortcut key. All other values are either inherited or the default is used.

In Figure 3.8 a poem ([Ada79]) is shown as marked-up in SGML, and in Figure 3.9 the same poem is shown using the semantic definitions.

## 3.7 Calculating environments

There are two different instantiations of an environment. The *resolved* environment is an environment where each characteristic is defined and its value is absolute. Only a resolved environment can be used to determine the layout and formatting of elements. Environments which have characteristics with relative values, or characteristics whose value is *inherit*, are termed to be *relative* environments. These environments cannot be used to format elements, and have to be resolved first by looking up the values in a resolved environment, resulting in another resolved environment.

When an element is created, the system calculates its own copy of the environment. This calculation involves the combination of several styles and environments. The default environment, either the built-in one or a user defined one, serves as a foundation. To this foundation the element applies its own tagstyle, evaluating any included style definitions, *when*- and *if*-clauses to create its relative environment. Earlier, the semantic actions corresponding to the parent of this element had created an environment by evaluating any

```

Scheme : "display"

Style for: poem
  Puttext : Before "Start of Poem"
           After  "End of Poem"

Style for: title
  Use environment: "heading"
  Labels : End    : ""

Style for: author
  Use environment: "bold-text"
  Puttext: Before "Author:"
    Font name   : "cmr10"
    Font style  : normal
  end Puttext

Style for: stanza
  Use environment : "quote"
  Left indent    : 5 spaces
  When IPRECED is NULL
    First indent : 3 spaces
    Prespace     : 2 lines
  Otherwise
    Prespace     : 1 line
  Labels : End   : ""

Style for: line
  Labels : Short : "L"

```

Figure 3.6: Tag styles for a simple poem

```

<!DOCTYPE poem [
<!ELEMENT poem   - - (title, author, stanza+)    >
<!ELEMENT title  - 0 (#PCDATA)                    >
<!ELEMENT author - 0 (#PCDATA)                    >
<!ELEMENT stanza - 0 (line)+                      >
<!ELEMENT line   - 0 (#PCDATA)                    >
]>

```

Figure 3.7: SGML document type definition for a simple poem

```

<poem>
<title>Poem
<author>Prostetnic Vagon Jeltz
<stanza>
<line>Oh freddled gruntbuggly thy micturations are to me
<line>As plurdled gabbleblotchits on a lurgid bee.
<stanza>
<line>Groop I implore thee, my foonting turlingdromes.
<line>And hooptiously drangle me with crinkly bindlewurdles,
<line>Or I will rend thee in the gobberwarts with my blurglecruncheon,
see if I don't!
</poem>

```

Figure 3.8: A poem marked up in SGML.

Start of Poem

Poem

Author:Prostetnic Vagon Jeltz

*'Oh freddled gruntbuggly thy micturations are to me  
As plurdled gabbleblotchits on a lurgid bee.'*

*'Groop I implore thee, my foonting turlingdromes.  
And hooptiously drangle me with crinkly bindlewurdles,  
Or I will rend thee in the gobberwarts with my blurglecruncheon, see if I don't!'*

End of Poem

Figure 3.9: Poem displayed using semantic definitions



relevant *for*-clauses and applying them to its own resolved environment. The semantic actions of the child then creates its own resolved environment by applying its calculated relative environment to the environment its parent created. The tagstyle defined for each child determines which characteristics are inherited and which not, and it is possible to inherit all characteristics, or none at all.

There are a few exceptions to these calculations. The root element has no explicit parent, but the root base environment (default or user supplied) is used as its implicit parent. The root element thus uses the root base environment to calculate its own resolved style. Furthermore, if a parent is *translucent*, it is bypassed by the child which instead searches up the document tree until it finds a non-translucent ancestor. This search is guaranteed to succeed as the root base environment is never translucent.

When elements are created or deleted it is often necessary to recalculate the environments, as the context of an element may have changed. When an element and its environment are created the conditional **when** and **if** statements which succeeded are saved explicitly with the environment of the element. If changes are made to the document through deletions or insertions these conditional statements are re-evaluated. Only if any of the conditional statements fail is the environment of the element recalculated. As the environments of the children of an element are very dependent on the parent's environment, their environment is also recalculated if the environment of the parent has changed.

If for an element the only conditional statements which succeeded are the **otherwise** or **else** conditions, then the environment of the element is also recalculated, as the context may have changed so that one of the **when** or **if** statements now evaluates to true.

## Chapter 4

# Rita+ System Overview

As with Rita, the Rita+ system consists of two different components: the Class Generator and the Editor. The Class Generator (RitaCG) compiles an SGML document type definition (DTD) into an intermediate file, which is subsequently loaded by the Editor. The Editor also loads in a semantic definition file, and can load and save user files marked up in SGML. If the semantic definition file provides more than one scheme, it is also possible to *export* documents in any of these schemes. User files are created according to the structure defined in the DTD and are displayed according to the semantic actions defined in the semantic definition file. Figure 4.1 shows a diagrammatic overview of the system. The user interface of the Editor remains the same as that of Rita, as shown in Figure 2.1.

### 4.1 Document creation and incomplete documents

As mentioned in Section 2.3, in the original Rita system documents had to be created in a front to back fashion. In other words, documents within Rita had to be created in the order defined by the class description.

In the system designed by [Smi87] the user was allowed to create documents in an almost arbitrary manner. Document creation may be *almost* arbitrary, as the user can never create documents which are illegal. Documents are termed to be legal if the partial (or incomplete) document has its elements in the correct order. For example, using the following regular expression:

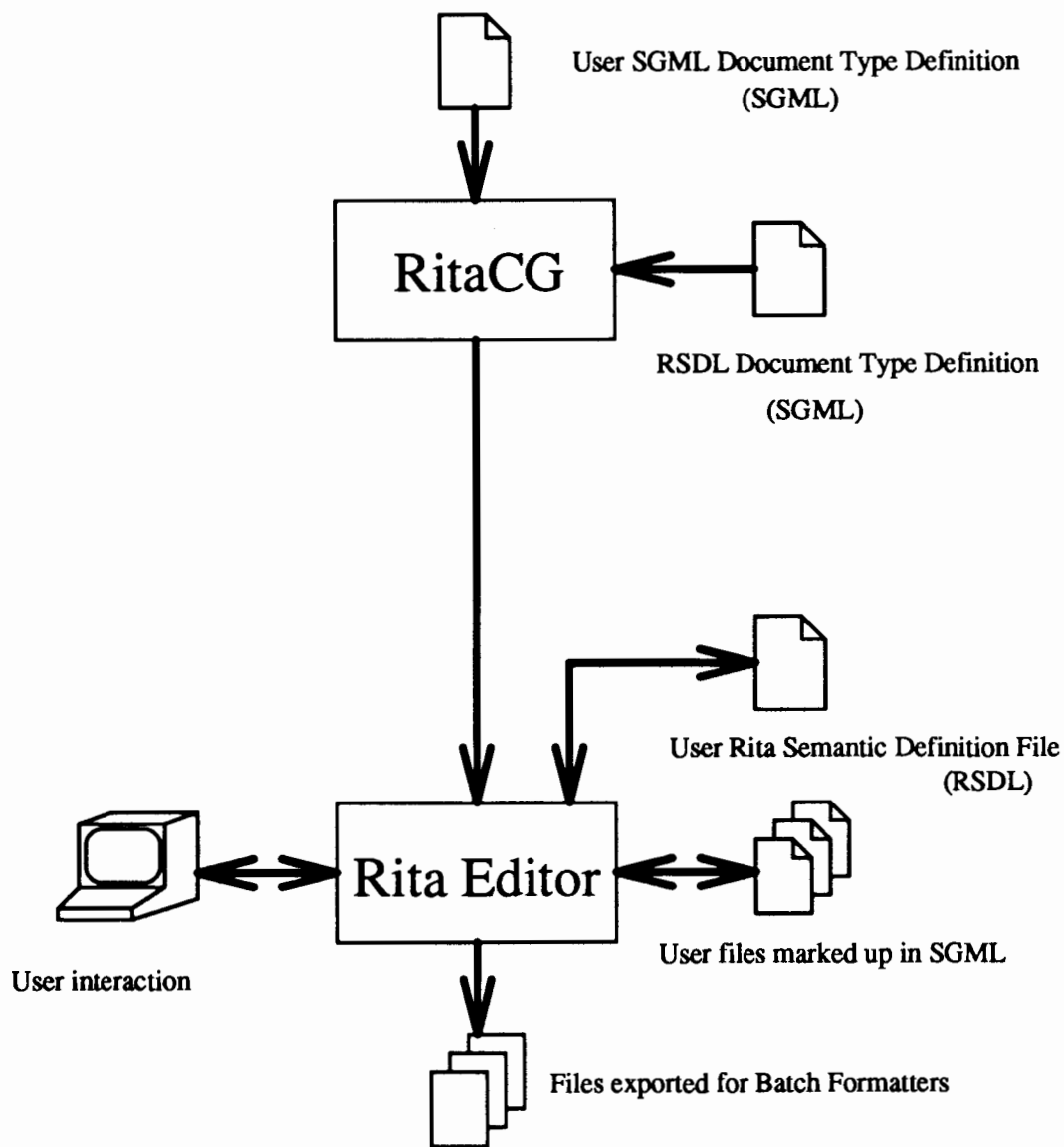


Figure 4.1: Rita+ System Overview

`document = heading body+ close`

A document regarded as legal would be `heading close`, as the elements are in order. By inserting element `body` after the `heading` the document would be complete. In contrast, it would not be legal to create the document `close body heading`, as it would not be possible to create a legal document by adding elements. A legal partial document is termed to be *sub-sequence* incomplete.

The Rita system did not support sub-sequence incompleteness, but this has been re-introduced in Rita+. Rita+ also helps the user to complete a document, if necessary, by

marking those items in the menu which are required to complete the current sub-sequence in the document using the least number of insertions.

## 4.2 Menu creation in Rita+

The Class Generator converts the regular expressions describing the document structure into corresponding deterministic finite state automata (DFA), which are saved as state transition tables in the intermediate file. The original Rita and Rita+ both read in these tables, but Rita+ then proceeds to convert each of the DFA's into a non-deterministic finite state automaton (NFA) by adding the  $\epsilon$  (null) symbol to each transition. This NFA is then converted according to the algorithm in [ASU86] to a corresponding DFA, which, in order to differentiate it from the original DFA, will be called the  $\epsilon$ -DFA. The DFA for the regular expression  $S = b^* a (a \mid b)^* c (a \mid b)^*$  is shown in Figure 2.7, while the  $\epsilon$ -DFA is shown in Figure 4.2.

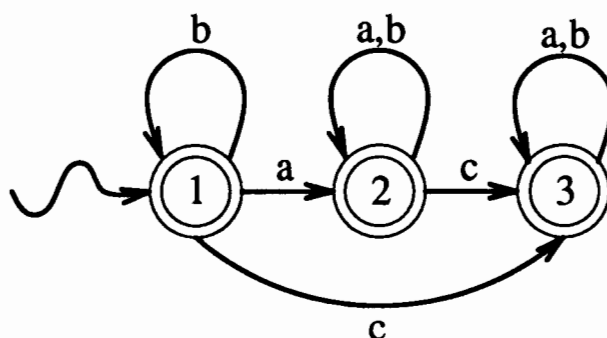


Figure 4.2:  $\epsilon$ -DFA created from the DFA

The reason for this conversion is that while Rita calculates its menus according to the DFA, Rita+ calculates its menus according to the  $\epsilon$ -DFA. The result of this difference can be seen immediately. With Rita, using insert after, on state 1 in Figure 2.7, the menu would consist only out of the elements a and b, whereas with Rita+, on state 1 in Figure 4.2, the menu would consist out of the elements a, b and c.

Thus, by using the DFA the user is restricted to creating documents in a sequential fashion, with only tail-sequence incompleteness. This restriction is removed in Rita+, and as the  $\epsilon$ -DFA is derived from the original DFA, documents can never be in an illegal state.

By allowing the user to create sub-sequence incomplete documents, problems may occur

<b>author</b>	<b>volume</b>
<b>title</b>	<b>number</b>
<b>journal</b>	<b>pages</b>
<b>year</b>	<b>note</b>

Table 4.1: BibTeX fields for an article entry.

if the user saves this file, as it would be nearly impossible to re-load this file into Rita+. Rita+ parses files read in according to the DFA, and while it would be possible to use the  $\epsilon$ -DFA to parse a file, ambiguities could occur, and furthermore other SGML parsers would not be able to process this file. Rita+ does allow the user to save a file in an incomplete state, but warns the user that the file is incomplete, and indicates which required sub-elements are missing.

Also, although the user may be aware of this problem, the user may not know how to complete a document, as the user may not know which elements are required and which are not. For example, bibliography entries in BibTeX require that the user supply for each entry a set of field values, some of which are optional and some of which are required [Lam86]. The problem is compounded in this case as the fields vary depending on the bibliographic entry. For example, Table 4.1 shows the fields that may appear for an article. From this table it is not obvious that the fields on the left are required, while the fields on the right are not. Rita+ solves this problem by aiding the user and marking those elements in the insert menu which would complete the subsection of the document with the least number of insertions.

#### 4.2.1 Menu marking

The menu marking calculation is a two-stage process [Smi87]. The first stage calculates the ways which would complete the current subsection of the document with the minimum number of insertions. As there may be several ways of completing this subsection, the result is a set of strings of equal length. For example, if the input string is *aba*, using the same regular expression *S* as before, then the result of this stage is

$$\left\{ \begin{array}{cccc} a & b & a & c \\ a & b & c & a \\ a & c & b & a \end{array} \right\}$$

The second stage uses this set of strings, the input language (the set of all elements defined

for the regular expression), the position in the string and the menu for this position as calculated using the  $\varepsilon$ -DFA. The result of this stage is a menu with those elements marked which would complete the input string in the minimum number of insertions. In Figure 4.3 the square brackets denote the menus with the marked elements underlined for each position in the string. In Rita+ itself this stage is only performed once for the current position in the input string.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} a \begin{bmatrix} a \\ b \\ \underline{c} \end{bmatrix} b \begin{bmatrix} a \\ b \\ \underline{c} \end{bmatrix} a \begin{bmatrix} a \\ b \\ \underline{c} \end{bmatrix}$$

Figure 4.3: Marked menus

As there can be several elements marked, as there may be several ways of completing the document, the menu marking has to be done each time the menu is calculated, as the set of minimal strings may change depending on which element has been inserted into the string.

Completing a document by only selecting marked elements does not guarantee that the whole document can be completed in a minimum number of insertions, but only the subsection corresponding to the expression for the current element.

## 4.2.2 Exceptions

Exceptions were included in the language designed by [Smi87], but were not included in the language used by Rita. Rita+ supports exceptions as SGML defines these.

Exceptions influence menu calculation, and once the menu has been calculated and marked, exceptions are dealt with. Included elements are added to the menu. These elements are never marked, as they are optional. Excluded elements are removed from the menu, even if the element is marked. If an element is both included and excluded, the exclusion dominates and the element is removed from the menu.

## 4.3 Using Rita+

To use Rita+ the Rita+ executable and a DTD (in the form of an intermediate file) are required. It is not necessary to specify a semantic file as there are built-in default

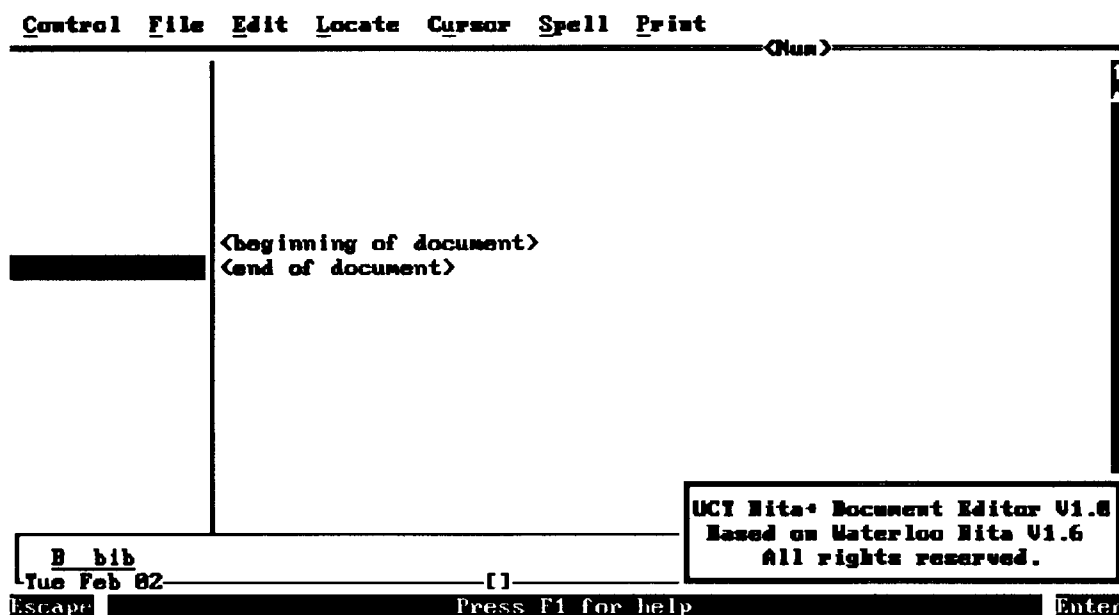


Figure 4.4: Starting up Rita+

actions. Each user file saved with Rita+ contains information as to which DTD and which semantic file are required for the file, and these are loaded automatically when the user file is loaded. If a new SGML marked-up file is loaded into the Editor the user can specify on the command line which DTD to use, and possibly which RSD file to use. If no DTD is specified a default DTD is loaded. Once a user file is saved from the Editor the first line in the document is a comment containing the DTD and RSD file information.

In Figure 4.4 Rita+ has just been started up, using the DTD and semantic file which are shown in Appendix C, Figures C.1 and C.2 respectively. As can be seen in Figure 4.4 the structure window is empty, as is the edit window, which only displays some default text as defined in Rita+. The dynamic menu shows the only possible entry, *Bib*, which, as it is underlined, is required.

Figure 4.5 shows the Editor after the item *article*, and its components *key* and *title* have been selected, and some text has been typed in. The cursor is currently on the *title* element, and the dynamic menu shows the items *author* and *note*. The *author* element is required, whereas the *note* element is an inclusion, which can be inserted (almost) anywhere, but is optional.

In the edit window it can be seen that both the *key* and *title* elements are displayed in bold text. In the semantic file shown in the appendix it can be seen that for the *title* element

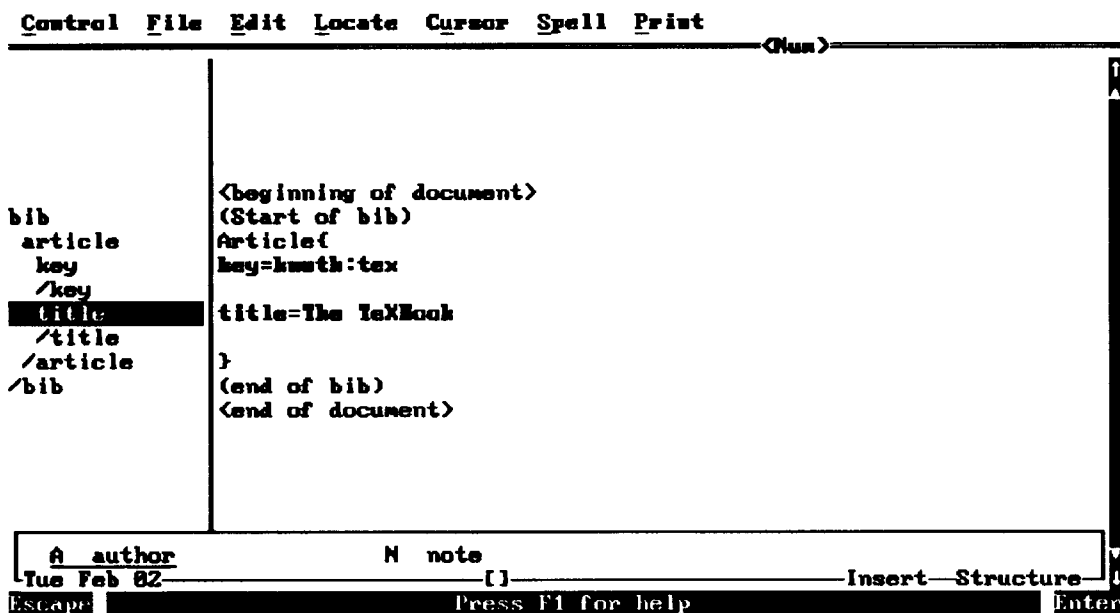


Figure 4.5: After inserting a key and a title

the font style changes according to its environment. Thus, if the immediate predecessor of this element is the *key* element, the font style is *bold*, but when it is the *author* element, the font style is *underline*.

In Figure 4.6 the *author* element is inserted, and as can be seen the font style of the *title* element has changed. The font style for *author* is given as *italics*, but as the PC display cannot display italicized text, it is shown as bold-underline.

In Figure 4.7 a *note* is added to the documents. Notes can be inserted almost anywhere. The note semantic definition is very simple; most of the characteristics are taken from the default style, shown in Appendix B. Thus, if a note is included within a *key* element the text would be bold, while within a *title* element the text would be underlined. As shown in Figure 4.7 the *note* is a child of the *article* element, and as its font style is plain, the font style for the note is plain as well.

By inserting the *note* before the *title* element, none of the contextual checks of the *title* style succeed, and as there is no explicit *otherwise* given, the default style is used. The result is that the font style becomes *plain*.



Control File Edit Locate Cursor Spell Print		(Num)
bib	<beginning of document> (Start of bib)	
article	Article{	
key	key=knuth:tex	
/key		
author	author=B. Knuth	
/author		
title	title=The TeXBook	
/title		
/article	}	
/bib	(end of bib) <end of document>	
Tue Feb 82 [ ] Insert Character		
Escape Press F1 for help Enter		

Figure 4.6: Adding an author

Control File Edit Locate Cursor Spell Print		(Num)
bib	<beginning of document> (Start of bib)	
article	Article{	
key	key=knuth:tex	
/key		
author	author=B. Knuth	
/author		
note	NOTE:Remember: first name is Donald	
/note		
title	title=The TeXBook	
/title		
/article	}	
/bib	(end of bib) <end of document>	
Tue Feb 82 [ ] Insert Character		
Escape Press F1 for help Enter		

Figure 4.7: Adding a note

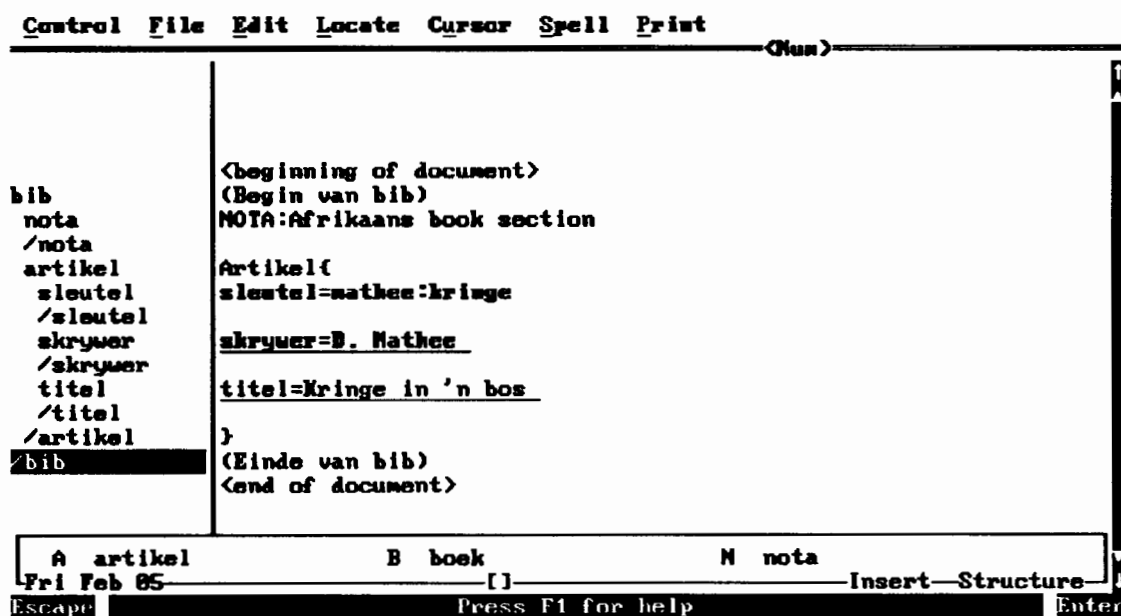


Figure 4.8: Using Afrikaans semantic definitions

## 4.4 Using the Rita Semantic Definition Language

Semantic files (or RSD files) are created using Rita+ itself. The definition of the RSDL language is represented using an SGML DTD, shown in Appendix D. RSD files are saved as documents marked up in SGML. Most of the information in such a file is contained within the structure tags, but information such as font name is saved as user text, as the interpretation of this field depends on the machine running Rita+. An SGML marked-up RSD file is shown in Figure C.3 in Appendix C, which corresponds to Figure C.2.

As RSD files are like any other user files, a semantic file can be created as for the RSDL document type definition as well, as shown in Appendix E, which shows the RSD on which the layout of all the RSD files shown in this thesis are based.

Rita+ uses RSD files as follows: once a DTD and the RSD files have been loaded, the element names are compared to the tag style names and associated. For elements which have no corresponding tag style defined, the default is used, and tag styles with no corresponding element are ignored. This process of association is performed each time a new RSD file is loaded, and it is thus possible to have completely different RSD files for the same DTD. In Figure 4.8 the same DTD is used as previously, but a different semantic file is used which caters for Afrikaans users.

By selecting the menu item *Control* and selecting item *Environment* Rita+ will display information regarding the current file name, which DTD is used and which semantic file is used.

## Chapter 5

# Changing Rita into Rita+, a critical analysis.

Rita is an established and working system based on G. de V. Smit's doctoral thesis [Smi87]. In the change from Rita to Rita+, not only did the *aim* of the system change, but also the *languages* to represent document structure and document semantics. This chapter will highlight some of the reasons for these changes, and will explain why it would be better to re-write the system rather than, as done for Rita+, modify the existing system.

### 5.1 Why the change to SGML and RSDL?

It was decided that the aim of the Rita system should change from being a front end to batch formatters to being an SGML editor. The system would remain a structured document editor, but it should now be able to read SGML DTD's, and the user should be able to create and modify documents marked up in SGML, and to display these documents in a manner determined by the user.

The original Class Description Language (CDL) which described document structure and document semantics was eliminated and replaced by SGML for document structure and RSDL for document semantics.

The Rita system is to a large extent GML based. Rita can read in files marked-up in GML, but it needs a special *scheme* for GML for it to be able to save files in GML format.

Not having a standard built-in method to save files was an advantage, as one version of the Rita editor was modified such that it could read in files marked-up in SGML as well. Adding an SGML scheme to the class description allowed the user to save files in SGML format. The disadvantage of not having a standard method for saving is that the user *always* has to supply an output scheme. Rita+, being fully SGML based, saves and loads files only in SGML, but, as with Rita, it is possible to export documents using other schemes.

The version of Rita which saves and loads files marked up in SGML files could now be used to process SGML document, except that CDL remained the same. This means that SGML document type definitions have to be rewritten as class descriptions. CDL uses regular expressions to define structure, and except for inclusions and exclusions, each SGML regular expression operator has a corresponding one in the language. The original CDL as defined by [Smi87] did define exceptions, and one version of the Rita editor as well as Rita+ also support them. Both SGML and CDL define macros for the structure section, and both provide for tag minimization. Thus, it would be possible to convert an SGML document type definition to a corresponding class description without semantics with relatively little effort.

However, before the user can process any document, semantic actions have to be provided. The semantic action section is an integral part of a class description, and it is necessary to define some form of semantic action, at least a display scheme and either a GML or SGML scheme for saving files. Semantic actions also have to be defined for each element in the document structure. This means that a certain amount of work has to be done even before a file can be viewed. Furthermore, before a class description can be used it has to be compiled by the Class Generator. Creating a semantic description for a file can thus become a tedious process. Each modification requires a re-compile and the Editor has to be re-started with the changed class description. It is thus not possible to quickly assign semantic actions to a file.

The language Rita used was only a subset of the language defined by [Smi87], making the language less expressive than intended. An effort was made to correct these problems, by modifying the language to correspond to the original language as proposed by [Smi87]. This was achieved by introducing proper variables and parameters, attributes, exception handling and adding several missing semantic unparsing commands.

But there still exist several problems with the semantic section of the class language, even with the modifications made.

### 5.1.1 Problems with CDL

The main problem with CDL is that it is very verbose. There are no macro constructs for semantics, only for structure. This means that it is necessary to specify semantic actions for each element, even if some of these elements behave similarly.

The language was also designed with only unsophisticated text screens in mind, so for example horizontal spacing is specified in characters, vertical spacing in lines, which is inadequate for proportional font bit mapped display systems. It is not possible to change font sizes, as fonts are limited to those provided by text screen PC's, and the range of font styles is very limited. The new system is intended to run on a variety of architectures, with a variety of display capabilities.

In Rita, semantic actions are specified in an "on transition" manner. The semantic actions are divided into those that have to be executed *before* any transition, those that have to be executed *after* all transitions and possibly those that have to be executed before a specific transition. The structure section of the semantic file is converted from a set of regular expression productions to a set of equivalent state transition tables corresponding to finite state automata, so the underlying mechanism of creating a document is to "move" from one document element to the next by means of a transition to that element. Semantic actions are performed by executing a set of actions, the *unparsing sequence*, on transitions. This means that to add semantic actions, the user has to understand the document structure, and the concept of finite state automata, to ensure that the semantic actions produce the right effect.

The structure of the language itself resulted in information being very distributed. Formatting effects could be set and remained in effect until reset or cleared. This set of semantic actions, essentially the environment, was kept globally. Each child of an element inherited this global environment, and had to reset it individually whenever required, and set it back to the original value afterwards. This could easily cause problems if the same element was present in several productions, and an element could have different parents in different sections of the document. The programmer of a class description thus had to

take care to ensure that there were no “dangling” environment settings.

### 5.1.2 Changing semantic languages

With the introduction of SGML to describe the document structure, the system had to be changed extensively, and rather than using the current inadequate language, it was decided to design a new language instead, a language which could cope with the changing scope and aim of the system. This new language had to be less verbose, which required the introduction of macros. Furthermore information would have to be kept where it was relevant and not distributed as before. Also, the language should be architecture independent, yet still provide all the functionality of the original class language. The resulting language was the *Rita Semantic Definition Language* or *RSDL* [Smi92].

RSDL differs in most aspects from the original language, not only in appearance but also in the underlying philosophy. The concept of *environments* is now the most important aspect. While with CDL semantic actions are a sequence of events, with RSDL they are a set of *characteristics*, all of which are defined for each element, and which together form the environment for that element. As default characteristics are defined it is not necessary for the user to specify each one explicitly, or for each element. Each environment is also local to that element. However, the child may inherit any characteristics of its parent's environment if it wants. If several children of an element are to be given a specific environment different from the element's current environment, this can be specified explicitly in the environment of the element, and this localizes information and avoids unnecessary duplication. Control over the current environment, however, always remains with the child, rather than with the parent as was the case with CDL, and an element can change the current environment completely, yet not affect any other elements except possibly any descendants, which can however completely ignore the environment of the parent. This means that the user can assign semantic actions to a specific element quickly, without having to be concerned to much about its context, and thus have to know the structure of the document.

### 5.1.3 Example of using RSDL and CDL

RSDL allows the user to specify information where it is required. CDL on the other hand distributes information in such a way that it is often difficult to understand the semantic actions. For example, consider the following structure definition:

```
book      = title, chapter+
chapter   = title, paragraph+
title     = TEXT
paragraph = TEXT
```

The user decides that the appearance of titles should be changed, to make it look more realistic. There are a few considerations: a title for a book has to be displayed in a completely different way than the title of a chapter. Book titles are generally set in a larger font size and possibly in a different font and font style. Creating semantic actions for headings using RSDL is not a problem. Following is one solution:

```
Style for: title
When Parent is "book"
  Font Name      : "Times-Roman"
  Font Style     : Bold
  Font Size      : 25pt
  Justification  : Centered
When Parent is "chapter"
  Font Name      : "Helvetica"
  Font Style     : Bold
  Font Size      : 15pt
  Justification  : Left Justified
```

As can be seen all the information required is saved in the tagstyle associated with **title**. The tagstyle uses *when*-statements to check its context, and acts according to which parent the element has.

This simple construct cannot be created using CDL. Following are the semantic actions as they would have to be specified using CDL.

```
book = title chapter+
[ /display/
  Method Standard()
    initial:
    title : @BookTitle(),
```



```

        final :
    endMethod
]

chapter = title paragraph+
[ /display/
    Method Standard()
        initial:
        title : @ChapterTitle(),
        final :
    endMethod
]

title = TEXT
[ /display/
    Method Standard()
        initial: ,
        final :
    endMethod

    Method BookTitle()
        initial: @bold @underline @center,
        final : @normal
    endMethod

    Method ChapterTitle()
        initial: @bold @left,
        final : @normal
    endMethod
]

```

The two *methods* defined for the CDL example correspond to the two *when* statements in the RSDL example, but it is immediately obvious that the CDL example is not only verbose, but also that information regarding the display of titles is now spread over the semantic definitions for three elements. To ensure that the **title** element uses the correct unparsing sequences each parent has to call it using a different method, and the **title** element has to define all the methods. The simple *when* clause as defined in the RSDL example is replaced here by “paths” from the different ancestors of **title**. As can also be seen it is not possible to change font size, so the text of the book title is underlined as well as bold, whereas the chapter title is only bold, as the text display of a Personal Computer does not support different font sizes.

Changing the structure can have far reaching effects within semantics using CDL. For

example, say chapters can be numbered. The regular expression production for chapter is changed, and a new production, called **heading** is introduced:

```
book      = title, chapter+
chapter   = heading, paragraph+
heading   = number?, title
number    = TEXT
title     = TEXT
paragraph = TEXT
```

With RSDL the changes are minimal:

```
Style for: title
When Parent is "book"
  Font Name      : "Times-Roman"
  Font Style     : Bold
  Font Size      : 25pt
  Justification  : Centered
When Parent of Parent is "chapter"
  Font Name      : "Helvetica"
  Font Style     : Bold
  Font Size      : 15pt
  Justification  : Left Justified
```

Only the conditional expression for the one *when*-statement had to be changed to refer to the parent's parent, thus bypassing the **heading** element. This is done as headings may also be used for sections, paragraphs, etc. The formatting of a title thus does not depend on **heading**, but on the parent of the heading. As the **Parent of Parent** construct may be seen as too "vague", it would also be possible to include an *if*-statement to ensure that the current **title** has a **heading** parent and a **chapter** grand-parent.

With the CDL the changes are similar, except that the result is much more verbose.

```
book = title chapter+
[ /display/
  Method Standard()
    initial:
      title : @BookTitle(),
    final :
  endMethod
]

chapter = heading paragraph+
[ /display/
```

```

        Method Standard()
            initial:
                heading: @ChapterTitle(),
            final :
        endMethod
    ]

heading = number? title
[ /display/
    Method Standard()
        initial: ,
        final :
    endMethod

    Method ChapterTitle()
        initial:
            title : @ChapterTitle(),
        final :
    endMethod
]

title = TEXT
[ /display/
    Method Standard()
        initial: ,
        final :
    endMethod

    Method BookTitle()
        initial: @bold @underline @center,
        final : @normal
    endMethod

    Method ChapterTitle()
        initial: @bold @left,
        final : @normal
    endMethod
]

```

As can be seen semantic actions for the element **heading** had to be created, which includes the need to create a special **ChapterTitle()** method to ensure that the correct semantic actions will be executed in the **title**.

Creating semantic actions using CDL can thus create a multitude of paths throughout the file, with a number of methods, making it hard to read and understand the semantic actions. Furthermore, changes cannot be made easily either to the structure or to the

semantics, as these changes can have far reaching effects.

#### 5.1.4 Using the RSDL language

The RSDL language is much more expressive than the original language. Some problems may occur as the semantic actions for an element for different output schemes (for example, display) are defined far apart, which may result in elements being omitted. Furthermore, as it is not possible to cross-check elements between the SGML DTD and RSD files automatically, it may be possible that elements are either omitted or ignored as they do not match any element in the DTD, which is easily possible, for example, due to spelling mistakes. However, errors such as these should be relatively easily to pick up if the results are not as expected. The Editor uses default environments for those elements which do not have an explicit environment defined.

Separating semantic actions for different schemes is not necessarily a deficiency in the language, as it may be more confusing to have all the schemes grouped together, making it difficult to modify a single scheme.

Another advantages of RSDL is that it exists as an SGML DTD, so the creation of semantic, or *RSD*, files can be done in Rita+ itself, and obviously it is possible to have an RSD file for the RSDL DTD as well. The RSD files themselves are just documents marked up in SGML, although most of the information is contained within the tags, and very little is text input by the user. The result of RSDL is that it is now possible to quickly assign semantic actions to a document, and to add these semantic actions in an incremental fashion.

## 5.2 Changing Rita into Rita+

With the new aim of the system, and its change from CDL to the SGML/RSDL combination, Rita required major modifications.

The class generator, RitaCG, which compiles the class description into an intermediate file now has to read an SGML DTD as input. The structure of the resulting intermediate file however remained the same, with the class generator adding simple default semantic actions, to ensure compatibility with Rita.

The code dealing with formatting on transitions has become obsolete, to be replaced with code which calculates the environment for each element. The difference in the underlying logic of these two methods made it impossible to implement all of the RSDL features in a satisfactory manner.

A few other areas had to be modified, such as the handling of exceptions as defined in SGML, as well as a few of the features mentioned in [Smi87] but not implemented in Rita, namely the subsequence incompleteness handling, and the menu calculation section.

### **5.3 The Rita+ system: implementation features**

Rita+ is written in the C language using the Watcom C compiler using a 80386sx based computer running MS-DOS V5.0. Additional tools used were the Watcom VIDEO debugger and custom software allowing remote debugging over both serial and parallel ports connected to a 80286 based computer. The user interface is written using a language and tools developed for Watcom by the University of Waterloo.

The change from Rita to Rita+ caused an increase in the source code related to Rita+ itself. This increase consists of about 15% new code and 12% modified code, resulting in a total increase of approximately 16%. The code related to the user interface only increased minimally. The size of the Rita+ executable is about 550 KBytes.

### **5.4 The Rita+ system: performance considerations**

Rita+ is a usable tool with adequate performance. There are some limitations in terms of speed and definite limitations in terms of document size. The largest document type definition used on a regular basis is the DTD for RSDL, used to create document semantic files. The file size limitations are because of Rita+ being built on top of Rita, which uses certain data structures which could not be eliminated. This also affected performance speed, and occasionally the system would hang when browsing quickly through the document as the internal system could not keep up the user interface. This problem, carried over from the Rita system, does not occur when browsing slowly, and is a problem of the user interface which was hardly touched at all for the implementation of Rita+. On

starting up Rita+ some time is required to convert the DFA's into NFA's and back into the  $\epsilon$ -DFA. This calculation time can be minimized by only using moderately sized regular expressions within the document type definition. Loading a fairly complex DTD such as the RSDL DTD caused a delay which is within the bounds of acceptability. Ideally this calculation should be done at compile time and pass the Editor the results in the intermediate file, but for practical reasons, namely keeping the structure of the intermediate file compatible with other versions of the Editor, this has not been done yet. The delay itself is small; the time required to start up Rita+ compared to Rita using similar sized files is about 10% longer. Calculating menus also causes a small delay, which however is only noticeable when comparing Rita+ with the original Rita system which used a far simpler menu calculation routine. The time delay however lies below half a second for a normal document type definition.

## 5.5 Arguments for a rewrite of Rita

With the introduction of RSDL, the handling of semantic actions has changed significantly, so that much of the code dedicated to this became obsolete. But Rita was designed with CDL in mind, and it was impossible to eliminate all the data structures and code used with the class language, as these are still partially used by the RSDL semantic actions.

Furthermore, much of the information in the intermediate file has become obsolete due to the introduction of RSDL, and the size of the file can become much smaller. It would now only be necessary to store the state-transition tables of each production, a symbol table and other information, such as tag minimization information. All the other information currently saved is present in the RSDL files.

The Rita system has evolved over several years, and its size and complexity make it impossible to determine exactly which code can be eliminated. Furthermore, several important sections such as menu calculation and semantic action handling have changed completely. Yet this new code still uses some of the older code, creating time and space inefficiencies.

A rewrite of Rita and creating a new system would be the only efficient way of eliminating these problems. Large sections of the original Rita code could still be used, such as the document tree handling with its "scaffolding" to access elements in a sequential fashion as

they are shown on the display, as well as the user interface which has hardly been touched at all.

## Chapter 6

# Conclusion

Rita+ allows the user to manipulate SGML documents in an interactive, yet structurally correct manner. The display of the document can be changed quickly using the Rita Semantic Definition Language, allowing users the freedom to change the appearance of the document according to their tastes and their idea of how a document should look like. With Rita+ even the casual user can, with little training, create SGML documents without having to understand the details of the document structure or having to know much about SGML. Furthermore, Rita+ does not bind the user in creating documents in a sequential, front to back, fashion, but rather allows the user to create documents in an almost arbitrary manner, yet still provides the user with an indication as to how to complete the document using a minimum of effort.

The introduction of SGML, an international standard for document markup and exchange, did not affect document structure handling of the system to a great extent. The structure definition of both SGML and the Class Description Language are essentially the same. As SGML provides more features than the Class Description Language used in Rita, it is well suited to describe document structure. However the introduction of SGML meant that semantic actions could not be specified anymore using the Class Descriptions, and semantic actions now had to be specified in a different way. The original Class Description Language was found to be inadequate for use with SGML, so a new language was created. The new semantic language, RSDL, allows quick assignment of semantic actions to a document as it is possible to assign semantic actions to individual elements whenever required. The user can thus create a semantic file incrementally. RSDL is also an improvement



over the Class Description language in that it stores information where it is needed, which contrasts sharply with the distribution of information found in a typical class description. This feature not only makes it easier to understand the semantic actions for a document, but also allows for quicker creation of a semantic file. Default actions ensure that document elements can be displayed, even though in a somewhat crude manner. This flexibility was not possible with the original system, where the user had to create a fairly complete class description before any document can be manipulated.

The use of environments to store semantic actions in RSDL differed from the method used with the Class Description Language, and this required extensive modifications to Rita. Issues such as calculating environments on document element creation, as well as recalculations of environments on insertions and deletions of document elements had to be dealt with. However, most of the display level routines of semantic actions were inherited from the Rita implementation, and only a few semantic action routines required modifications.

The implementation of the Rita+ system shows that RSDL is usable for what it is intended, namely adding semantics to an SGML document quickly. Also, the incomplete document support, which was missed in the implementation of Rita, gives the user more freedom in creating documents. It also removed the restrictions Rita imposed on the user by only allowing tail sequence incomplete documents, yet this was achieved without adding excess overhead to the overall performance of the system, even though the menu calculation routines changed extensively. In conclusion, Rita+, as an SGML document manipulating system, is thus more versatile, flexible and arguably better system than the original Rita.

## 6.1 Further work

Rita+ is, however, not complete. Rita+ was implemented on a system using only a text based display, and thus the advantages of multiple font names, sizes and styles is lost. Within the definition of RSDL the conditional statement options are only a subset of what is possible, but these are easy to implement.

As with all preceding systems Rita+ does not support the creation of tables, figures and mathematical equations. Although tables and mathematical expressions possess struc-

ture, it is not easy to display this structure using the Rita+ interface, as Rita+ displays structure vertically, while mathematical expressions would require the horizontal display of structure. Similar problems are encountered when embedding small document elements, such as highlighted words within a line of text. Although this problem has been solved to some extent by introducing a “verbose” mode in the Editor which splits the line, leaving the highlighted word on a line by itself, together with its corresponding tag, the solution is not ideal. Compared to tables and mathematical expressions figures have little or no structure, and thus pose a problem not only with the display of the structure, but also in the creation of the figure, as the structure is generally not sequential.

Rita+ allows users to create semantic definition files using the Rita+ Editor itself. This means however that the RSDL DTD has to be loaded, the RSD file modified and subsequently the user file has to be reloaded. For small changes to the semantic actions this is excessive work, and it should be possible to change semantic actions even faster. This could be done by selecting the element whose environment should be changed, and allow the user to modify the characteristics in an online fashion. These changes could then either be saved in the original RSD file, or possibly only the changes should be saved in a file. In this manner the original RSD file is not changed, yet each individual user can modify the display of their documents.

## Appendix A

# Standard Generalized Markup Language (SGML)

SGML, an international standard for document interchange [ISO86], is a language to describe documents in a declarative way. With SGML, documents are interchanged by providing both the marked-up document and a document type definition containing definitions of the kind of mark-up used. SGML is not a document formatter [Bar89]. With SGML, markup is used to describe the structure of the document and how sections of the document relate to each other, and not how the document should appear. Thus SGML differs from procedural formatters like  $\text{\TeX}$  where the user specifies how each section of the document has to be formatted.

### A.1 Marking up documents

A document is marked up in SGML by demarcating document components using start and end tags according to the specification contained in some document type definition. An example of a marked-up document is shown in Figure A.1, which is marked up according to the document type definition of Figure A.2.

The format of start tags is `<tag name>` and the format for end tags is `</tag name>`. Tags may be omitted if the document remains unambiguous and if the document type definition allows for it. For example, a start tag for a paragraph implies an end tag for

```

<document ident=markup>
  <heading>
    This is the heading of the document
  <body>
    <paragraph>
      The end of the heading is unambiguously terminated by the start of
      the body. The indentation is incidental, and its only purpose is to
      make this example more readable. Tags may in fact occur anywhere
      within the document.
    </paragraph>
    <example>
      <paragraph>
        A bit more text, this time in an example
      </example>
    </body>
  <close>
    <paragraph>
      And a close to finish it all
    </close>
</document>

```

Figure A.1: A small SGML marked-up document

the previous paragraph and thus the end tag may be omitted. In Figure A.1 the second paragraph does not have a close tag, as the close tag for `example` implies it. The end tag for `heading` is implied by the start tag of `body`. An empty end tag (`</>`) matches the most recent start tag.

Start tags may also contain attribute values which are used to attach properties to an element. Attributes can be used if the corresponding element in the document type definition provides for attributes. In Figure A.1 the `document` element has an attribute, `ident`, which is set to the string “markup”.

To parse a document marked up with SGML the document and its corresponding document type definition, which define the tags and their relationship, have to be provided.

## A.2 Document Type Definitions

A document type definition (DTD) describes the mark-up of a set or class of documents. Each DTD consists of several sections. These sections are the *entity* declarations, the

```

<!-- Small SGML DTD for a simple document.                -->
<!-- Entity declarations                                    -->
<!--                                                        -->
<!ENTITY % content "heading, body, close"                  >
<!--                                                        -->
<!-- Element declarations                                    -->
<!-- Element Name   Tag min.   Regular expression   Exceptions -->
<!-- -----      - - - - -   -----              - - - - - -->
<!ELEMENT document  - -       (%content;)             +(note)    >
<!ELEMENT heading   0 0       (#PCDATA)                >
<!ELEMENT body       - 0      ((paragraph | example)+) >
<!ELEMENT paragraph - 0      (#PCDATA)                >
<!ELEMENT example    - 0      (paragraph)              >
<!ELEMENT close      - -      (paragraph)              >
<!ELEMENT note       - -      (paragraph)              -(note)    >
<!--                                                        -->
<!-- Attribute definitions                                    -->
<!ATTLIST document  ident ID    #REQUIRED              >

```

Figure A.2: A small SGML DTD

*element* declarations, which define the document structure, *element attribute* declarations and comments. Figure A.2 shows a typical (if rather short) document type definition.

### A.2.1 Comments

Single line comments in SGML are delimited by `<!--` and `-->`, while comments within *entity* or *element* declarations are delimited simply by a double dash (`--`) before and after the comment. For example, in Figure A.2 the first three lines are comments.

### A.2.2 Defining document structure

The structure of a document is defined using regular expression productions. Each regular expression defines an *element*. The name of an element is used within user documents as the tagname.

The regular expression operators defined in SGML [Gol90] are the same as those defined for the Class Description Language, and are shown in Table 2.1. The only differences are syntactical, in that SGML separates sequential elements with a comma, and uses the `&` character instead of the `#` character as the and operator.

SGML allows the use of *exceptions* within regular expressions. Exceptions are used to either include or exclude elements from the content of an element and its children. For example, in Figure A.2 the element `note` is included in the expression of `document`, while it is excluded in the expression for `note` itself. This means that a note may appear anywhere in the document, except within notes themselves.

SGML defines a variety of terminal symbols, of which the most common one is `#PCDATA`, for *parsed character data* (i.e. plain text), and `EMPTY`, which denotes an element with empty content.

As mentioned before, tags may be omitted within the user document if the structure remains unambiguous. In the DTD in Figure A.2 the column marked `Tag min.` indicates which tags may be minimized, where the first character in the column represents the start tag, the second character the end tag. If the character is a dash then the tag is required, if however the character is an `O`, it is optional.

### A.2.3 Entities

Entities are named objects, which consist of an entity name and a text string. Whenever an entity name is encountered, the SGML parser replaces this name with the text string.

Entities can be used within DTD's to replace long, commonly occurring regular expressions, or within documents to replace long strings by a shortcut. For example, one could define an entity `SGML` to stand for *Standard Generalized Markup Language*. Entities are also useful to represent characters which are reserved for SGML, such as the angled brackets `<` and `>`. For example, the declaration `<!ENTITY lt "<" >`, allows the user to include left angle brackets by specifying `&lt;`. In Figure A.2 an entity, `content`, is declared and used in the `document` element.

### A.2.4 Attributes

Attributes allow a user to attach properties to an element. An attribute definition consists of the element name to which it is associated, an attribute name, an attribute type and a default value.

The attribute type can be one defined in SGML, such as *ID* for identifiers and *CDATA* for character data, or a list of legal values. The default value may be an indication as to whether the attribute is required, optional, whether it assumes the current value, or whether it uses a given value as default.

In the SGML DTD shown in Figure A.2 an attribute is defined for the *document* element. This attribute is of the type identifier and is required. The value of this attribute is set to “markup” as shown in Figure A.1.

## Appendix B

# Default style definitions

The default root base environment as shown in Figure B.1 is used as the basis environment for any document, while the default environment in Figure B.2 is used to ensure that all characteristic fields of an environment are defined. **Form**, **Labels** and **puttext** cannot be relative, and are thus defined within the default, and not the root default, to ensure that they are defined for each element.

```
Scheme      : "display"
Style for   : rootdefault
  Left Indent : 0
  Right Indent : 0
  First Indent : 0
  Line Length : 64 spaces
  Line Height : 1 line
  Prespace    : 0
  Postspace   : 0
  Fontname     : "Normal"
  Fontsize    : 10 pt
  Fontstyle    : Plain
  Justify      : Left
  Tabs        : Initial 0, Increment 8 spaces
```

Figure B.1: Default root base environment for PC.



```

Scheme      : "display"
Styledef    : "default"
  Left Indent : +0
  Right Indent : +0
  First Indent : +0
  Line Length : +0
  Line Height : +0
  Prespace    : +0
  Postspace   : +0
  Fontname     : Inherit
  Fontsize    : +0
  Fontstyle    : Inherit
  Form         : Block (smooth, smooth)
  Justify      : Inherit
  Tabs         : Inherit
  Labels       : Begin tagname
                  End   "/" tagname
                  Menu   tagname
                  Short  tagname
  Puttext      : Before ""
                  After ""

```

Figure B.2: Default environment for PC.

## Appendix C

# Sample DTD's and semantic files

Figures C.1, C.2 and C.3 are used as examples in Section 4.3.

Figure C.1 shows a sample SGML document type definition for a simple bibliography. The structure of the bibliography defined here is as follows: a **bib** consists out of a collection of entries which may be the elements **article** or **book**. Both the **article** and **book** elements consist out of the text fields **key**, **author** and **title**. It is possible to add **notes** anywhere, except within a **note** itself, as **note** is included in the top element, **bib**, but excluded in the **note** element itself.

Figure C.2 shows the document semantics in RSDL corresponding to the document type definition. The styles for **bib**, **article**, **book**, **key** and **author** simply display some text on the display. The style for **title** changes the style of the font; underline if the immediate preceding element is **author**, or bold if the preceding element is **key**

Figure C.3 shows how Rita+ actually stores the document semantic file shown in Figure C.2, namely marked up as an SGML file.

```

<!doctype bib [
<!-- Element Tag min Production Exceptions -->
<!ELEMENT bib - - (article | book)+ +(note) >
<!ELEMENT article - - (key, author, title) >
<!ELEMENT book - - (key, author, title) >
<!ELEMENT key - 0 (#PCDATA) >
<!ELEMENT author - 0 (#PCDATA) >
<!ELEMENT title - 0 (#PCDATA) >
<!ELEMENT note - - (#PCDATA) -(note) >
]>

```

Figure C.1: Sample SGML DTD.

```

Scheme      : "display"
Style for   : bib
Puttext     : Before "(Start of bib)"
              After  "(End of bib)"

```

```

Style for   : article
Puttext     : Before "Article{"
              After  "}"

```

```

Style for   : book
Puttext     : Before "Book{"
              After  "}"

```

```

Style for   : key
Fontstyle   : Bold
Puttext     : Before "key="

```

```

Style for   : author
Fontstyle   : Italics
Puttext     : Before "author="

```

```

Style for   : title
Puttext     : Before "title="
When IPRECEDED is "author"
  Fontstyle : Underline
When IPRECEDED is "key"
  Fontstyle : Bold

```

```

Style for   : Note
Puttext     : Before "NOTE:"

```

Figure C.2: Sample RSDL semantic definitions

```

<!--**RTA**RSDL **none** -->
<rsdl><scheme><ident>
DISPLAY</ident>
<tagstyle><tag>
bib</tag>
<envblok><character><puttext><before><text>
(Start of bib)</text></before>
<after><text>
(end of bib)</text></after></puttext></character></envblok></tagstyle>
<tagstyle><tag>
article</tag>
<envblok><character><puttext><before><text>
Article{</text></before>
<after><text>
}</text></after></puttext></character></envblok></tagstyle>
<tagstyle><tag>
book</tag>
<envblok><character><puttext><before><text>
Book{</text></before>
<after><text>
}</text></after></puttext></character></envblok></tagstyle>
<tagstyle><tag>
key</tag>
<envblok><character><styleP><fontstyl><bold>
</fontstyl></styleP>
<puttext><before><text>
key=</text></before></puttext></character></envblok></tagstyle>
<tagstyle><tag>
author</tag>
<envblok><character><styleP><fontstyl><italics>
</fontstyl></styleP>
<puttext><before><text>
author=</text></before></puttext></character></envblok></tagstyle>
<tagstyle><tag>
title</tag>
<envblok><character><puttext><before><text>
title=</text></before></puttext></character></envblok>
<when><context><iprecd><tag>
author</tag></context>
<envblok><character><styleP><fontstyl><underlin>
</fontstyl></styleP></character></envblok></when>
<when><context><iprecd><tag>
key</tag></context>
<envblok><character><styleP><fontstyl><bold>
</fontstyl></styleP></character></envblok></when></tagstyle>
<tagstyle><tag>
note</tag>
<envblok><character><puttext><before><text>
NOTE:</text></before></puttext></character></envblok></tagstyle></scheme>
</rsdl>

```

Figure C.3: Sample RSDL semantic definitions in SGML format

## Appendix D

# The Rita Semantic Language Document Type Definition

Following is the document type definition in SGML for the Rita Semantic Definition Language. This type definition can be used with Rita+ in conjunction with the semantic definition file shown in Appendix E to create semantic definition files.

```
<!-- ===== -->
<!-- RSDL specification. V1.4.2 -->
<!-- Created January 1993 by G. Zsilavec -->
<!-- Laboratory for Advanced Computing -->
<!-- Department of Computer Science -->
<!-- University of Cape Town -->
<!-- ===== -->

<!doctype rsdl [

<!-- Used to be macros, but made elements to increase performance -->

<!ELEMENT blocksp - - (indentL?, firstI?, indentR?, llength?, lheight?,
                        presp?, postsp?, tabs?) >
<!ELEMENT styleP - - (fontname?, fontsize?, fontstyl?, form?,
                        justify?, blocksp?) >
<!ELEMENT charact - - (transluc?, suppress?, styleP?, enum?, savetext?,
                        puttext?, ifclause?) >
<!ELEMENT envblok - - (envref?, charact?, forclaus*) >

<!-- Top Level Element Definitions -->

<!ELEMENT rsdl - - (scheme*) >
```

```

<!ELEMENT scheme - - (ident, styledef*, tagstyle*) >
<!ELEMENT styledef - - (ident, envref?, charact?, labels?) >
<!ELEMENT tagstyle - - (tag+, envblok?, (when+, otherwis?)?, labels?) >

<!-- Top level Component Element Definitions -->

<!ELEMENT ident 0 0 (#PCDATA) >
<!ELEMENT tag 0 0 (#PCDATA) >
<!ELEMENT when - - (context+, envblok?) >
<!ELEMENT otherwis - - (envblok?) >
<!ELEMENT context - - (( parent | ipreced | ifollow)+,
                        not?, (tag | null)) >
<!ELEMENT envref - 0 (#PCDATA) >
<!ELEMENT transluc - 0 EMPTY >
<!ELEMENT suppress - 0 EMPTY >
<!ELEMENT not - 0 EMPTY >
<!ELEMENT parent - 0 EMPTY >
<!ELEMENT ipreced - 0 EMPTY >
<!ELEMENT ifollow - 0 EMPTY >
<!ELEMENT null - 0 EMPTY >

<!-- For Clause -->

<!ELEMENT forclaus - - (tag+, envref?, charact?) >

<!-- Labels -->

<!ELEMENT labels - - (begin?, end?, menu?, short?) >
<!ELEMENT begin - - (text | tagname) >
<!ELEMENT end - - (text | tagname) >
<!ELEMENT menu - - (text | tagname) >
<!ELEMENT short - - (text | tagname) >
<!ELEMENT tagname - 0 EMPTY >

<!ELEMENT inherit - 0 EMPTY >

<!-- Block Spacing -->

<!ELEMENT indentL - - (select) -(ln) >
<!ELEMENT firstI - - (select) -(ln) >
<!ELEMENT indentR - - (select) -(ln) >
<!ELEMENT llength - - (select | window) -(ln) >
<!ELEMENT lheight - - (select) -(sp) >
<!ELEMENT presp - - (select) -(ln) >
<!ELEMENT postsp - - (select) -(ln) >
<!ELEMENT tabs - - ((select, select) | tablist | inherit) -(ln) >
<!ELEMENT tablist - - ( text, units )+ >
<!ELEMENT window - 0 EMPTY >

```

```

<!-- Style Parameters                                -->

<!ELEMENT fontname - - (text | inherit)              >
<!ELEMENT fontsize - - (select)                      -(ln | sp) >
<!ELEMENT fontstyl - - (plain | bold | italics | underlin | inherit) >
<!ELEMENT form      - - (block | inline | page | inherit) >
<!ELEMENT block     - - ((smooth | sticky | rough),
                        (smooth | sticky | rough))      >

<!ELEMENT justify   - - (left | right | both | centered | inherit) >
<!ELEMENT plain     - 0 EMPTY                                     >
<!ELEMENT bold      - 0 EMPTY                                     >
<!ELEMENT italics   - 0 EMPTY                                     >
<!ELEMENT underlin  - 0 EMPTY                                     >
<!ELEMENT inline    - 0 EMPTY                                     >
<!ELEMENT page      - 0 EMPTY                                     >
<!ELEMENT smooth    - 0 EMPTY                                     >
<!ELEMENT sticky    - 0 EMPTY                                     >
<!ELEMENT rough     - 0 EMPTY                                     >
<!ELEMENT left      - 0 EMPTY                                     >
<!ELEMENT right     - 0 EMPTY                                     >
<!ELEMENT both      - 0 EMPTY                                     >
<!ELEMENT centered  - 0 EMPTY                                     >

<!-- Characteristics                                -->

<!ELEMENT enum      - - (ident, initial?, inc?, within+)      >
<!ELEMENT initial   - 0 (#PCDATA)                               >
<!ELEMENT inc       - 0 (#PCDATA)                               >
<!ELEMENT within    - 0 (#PCDATA)                               >
<!ELEMENT puttext   - - ( (before?, after? ) | bothside )     >
<!ELEMENT before    - - ( (ident | text )*, styleP? )         >
<!ELEMENT after     - - ( (ident | text )*, styleP? )         >
<!ELEMENT bothside  - - ( (ident | text )*, styleP? )         >
<!ELEMENT savetext  - - (ident, (ident | text)* )             >

<!-- If Clause                                       -->

<!ELEMENT ifclause  - - (context, envref?, charact?, elseif*, else?) >
<!ELEMENT elseif    - - (context, envref?, charact?)           >
<!ELEMENT else      - - (envref?, charact?)                     >

<!-- Selection                                       -->

<!ELEMENT select    - 0 ((plus | minus)?, ((text, units) | zero)) >
<!ELEMENT plus      - 0 EMPTY                                     >
<!ELEMENT minus     - 0 EMPTY                                     >

```

<!ELEMENT zero	- 0 EMPTY	>
<!ELEMENT units	0 0 ( in   cm   pt   ln   sp )	>
<!ELEMENT in	- 0 EMPTY	>
<!ELEMENT cm	- 0 EMPTY	>
<!ELEMENT pt	- 0 EMPTY	>
<!ELEMENT ln	- 0 EMPTY	>
<!ELEMENT sp	- 0 EMPTY	>
<!ELEMENT text	- 0 (#PCDATA)	>
]>		



## Appendix E

# Rita Semantic Definition File for the Rita Semantic Language

Following is the RSD file corresponding to the RSDL DTD definition found in Appendix D.

**Scheme : "display"**

**Environment : base**

**Font name : "cmtt"**

**Font size : 10 pt**

**Font style : plain**

**Form : Block (smooth, smooth)**

**Justification : left**

**Style for : blocksp**

**Style for : styleP**

**Style for : charact**

**Style for : envblock**

**Style for : rsdl**

**Use environment "base"**

**Puttext : Before "(Start of Semantic File)"**

**After "(End of Semantic File)"**

**Style for : scheme**

**Puttext : Before "Scheme :"**

**Font style : bold**

**After ""**

Font style : bold

Style for : styledef  
 Puttext : Before "Environment :"  
           Font style : bold  
 For children "envref", "charact", "labels"  
           Left indent : +2 spaces

Style for : tagstyle  
 Puttext : Before "Style for :"  
           Font style : bold  
 For children "envblok", "when", "otherwis", "labels"  
           Left indent : +2 spaces

Style for : ident  
 When PARENT is "styledef"  
           Font Style : underline

Style for : tag  
 When PARENT is "tagstyle"  
           Font Style : underline  
 When PARENT is "context"  
     Puttext : Before ""  
               After ""  
 When PARENT is "forclaus"  
     Puttext : Before ""  
               After ""

Style for : when  
 Puttext : Before "When "  
           Font style : bold

Style for : otherwis  
 Puttext : Before "Otherwise"  
           Font style : bold

Style for : context  
 When IPRECED is "context"  
     Puttext : Before " and "  
               Font style : bold

Style for : envref  
 Puttext : Before "Use environment ""  
           Font style : bold  
     After ""  
           Font style : bold

Style for : transluc

Puttext : Before "Translucent"  
Font style : bold

Style for : suppress  
Puttext : Before "Suppress"  
Font style : bold

Style for : not  
Puttext : Before " NOT "

Style for : parent  
Font style : italics  
When IPRECED is NULL  
If IFOLLOW is NULL  
Puttext : Before " PARENT is "  
Else  
Puttext : Before " PARENT "  
Otherwise  
If IFOLLOW is NULL  
Puttext : Before " of PARENT is "  
Else  
Puttext : Before " of PARENT "

Style for : ipreced  
Font style : italics  
When IPRECED is NULL  
If IFOLLOW is NULL  
Puttext : Before " IPRECED is "  
Else  
Puttext : Before " IPRECED "  
Otherwise  
If IFOLLOW is NULL  
Puttext : Before " of IPRECED is "  
Else  
Puttext : Before " of IPRECED "

Style for : ifollow  
Font style : italics  
When IPRECED is NULL  
If IFOLLOW is NULL  
Puttext : Before " IFOLLOW is "  
Else  
Puttext : Before " IFOLLOW "  
Otherwise  
If IFOLLOW is NULL  
Puttext : Before " of IFOLLOW is "  
Else

Puttext : Before " of IFOLLOW "

Style for : null  
 Puttext : Before " NULL "  
     Font style : italics

Style for : forclaus  
 Puttext : Before "For children "  
     Font style : italics  
 For children "tag"  
   if IPRECED is "tag"  
     Puttext : Before ", ""  
   else  
     Puttext : Before ""  
     Puttext : After ""  
 For children "charact", "envref"  
   Left indent : +2 spaces

Style for : labels  
 Puttext : Before "Labels:"  
     Font style : bold

Style for : begin  
 Puttext : Before "Begin"  
     Font style : bold  
 When IPRECED is NOT NULL  
   Left Indent : +8 spaces

Style for : end  
 Puttext : Before "End"  
     Font style : bold  
 When IPRECED is NOT NULL  
   Left Indent : +8 spaces

Style for : menu  
 Puttext : Before "Menu"  
     Font style : bold  
 When IPRECED is NOT NULL  
   Left Indent : +8 spaces

Style for : short  
 Puttext : Before "Short"  
     Font style : bold  
 When IPRECED is NOT NULL  
   Left Indent : +8 spaces

Style for : tagname  
 Puttext : Before "tagname"

Font style : italics

Style for : inherit  
 Puttext : Before "inherit"  
 Font style : italics

Style for : indentL  
 Puttext : Before "Left indent :"  
 Font style : bold

Style for : indentR  
 Puttext : Before "Right indent :"  
 Font style : bold

Style for : firstI  
 Puttext : Before "First indent :"  
 Font style : bold

Style for : llength  
 Puttext : Before "Line length :"  
 Font style : bold

Style for : lheight  
 Puttext : Before "Line height :"  
 Font style : bold

Style for : presp  
 Puttext : Before "Prespace :"  
 Font style : bold

Style for : postsp  
 Puttext : Before "Postspace :"  
 Font style : bold

Style for : tabs  
 Puttext : Before "Tabs :"  
 Font style : bold  
 For children "select"  
 if IPRECED is "select"  
 Puttext : Before ",Increment:"  
 else  
 Puttext : Before "Initial:"

Style for : tablist  
 When IPRECED is NOT NULL  
 Puttext : Before ", "

Style for : window

Puttext : Before "window"  
Font style : italics

Style for : fontname  
Puttext : Before "Font name :"  
Font style : bold  
After ""  
Font style : bold

Style for : fontsize  
Puttext : Before "Font size :"  
Font style : bold

Style for : fontstyl  
Puttext : Before "Font style :"  
Font style : bold

Style for : form  
Puttext : Before "Form :"  
Font style : bold

Style for : block  
Puttext : Before "Block ("  
After ")"

Style for : justify  
Puttext : Before "Justification :"  
Font style : bold

Style for : plain  
Puttext : Before "plain"  
Font style : italics

Style for : bold  
Puttext : Before "bold"  
Font style : italics

Style for : italics  
Puttext : Before "italics"  
Font style : italics

Style for : underlin  
Puttext : Before "underline"  
Font style : italics

Style for : inline  
Puttext : Before "inline"  
Font style : italics

Style for : page  
Puttext : Before "page"  
Font style : italics

Style for : smooth  
Font style : italics  
When IPRECED is NULL  
Puttext : Before "smooth"  
Otherwise  
Puttext : Before ", smooth"

Style for : sticky  
Font style : italics  
When IPRECED is NULL  
Puttext : Before "sticky"  
Otherwise  
Puttext : Before ", sticky"

Style for : rough  
Font style : italics  
When IPRECED is NULL  
Puttext : Before "rough"  
Otherwise  
Puttext : Before ", rough"

Style for : left  
Puttext : Before "left"  
Font style : italics

Style for : right  
Puttext : Before "right"  
Font style : italics

Style for : both  
Puttext : Before "both"  
Font style : italics

Style for : centered  
Puttext : Before "centered"  
Font style : italics

Style for : enum  
Puttext : Before "Enum"  
Font style : bold

Style for : initial  
Puttext : Before "Initial:"

Font style : italics

Style for : inc  
 Puttext : Before "Increment:"  
 Font style : italics

Style for : within  
 When IPRECED is NULL  
 Puttext : Before "Within:"  
 Font style : italics  
 Otherwise  
 Puttext : Before ",,"

Style for : puttext  
 Puttext : Before "Puttext :"  
 Font style : bold

Style for : before  
 Puttext : Before "Before "  
 Font style : bold  
 For children "styleP"  
 Left indent : +13 spaces

Style for : after  
 Puttext : Before "After "  
 Font style : bold  
 When IPRECED is "before"  
 Left indent : +11 spaces  
 For children "styleP"  
 Left indent : +2 spaces  
 Otherwise  
 For children "styleP"  
 Left indent : +13 spaces

Style for : bothside  
 Puttext : Before "Both sides "  
 Font style : bold  
 For children "styleP"  
 Left indent : +13 spaces

Style for : savetext  
 Puttext : Before "Savetext: "  
 For children "ident"  
 If IPRECED is NULL  
 Puttext : Before "="

Style for : ifclause  
 Puttext : Before "If "



```

        Font style : bold
    After " endif"
        Font style : bold
    For children "charact"
        Left indent : +2 spaces

Style for : elseif
    Puttext : Before " else if "
        Font style : bold
    For children "charact"
        Left indent : +2 spaces

Style for : else
    Puttext : Before " else "
        Font style : bold
    For children "charact"
        Left indent : +2 spaces

Style for : select

Style for : plus
    Puttext : Before "+"

Style for : minus
    Puttext : Before "-"

Style for : zero
    Puttext : Before "0"

Style for : units

Style for : in
    Puttext : Before "in"
        Font style : italics

Style for : cm
    Puttext : Before "cm"
        Font style : italics

Style for : pt
    Puttext : Before "pt"
        Font style : italics

Style for : ln
    Puttext : Before "lines"
        Font style : italics

Style for : sp

```

**Puttext : Before "spaces"**  
**Font style : italics**  
**Style for : text**

# Bibliography

- [Ada79] D. Adams. *The Hitch Hiker's Guide to the Galaxy*. Pan Books, Cavaye Place, London SW10 9PG, 1979.
- [Ass88a] Association of American Publishers. *Author's guide to electronic manuscript preparation and markup*, version 2.0 revised edition, 1988.
- [Ass88b] Association of American Publishers. *Reference manual on electronic manuscript preparation and markup*, version 2.0 revised edition, 1988.
- [Ass88c] Association of American Publishers. *Standard for electronic manuscript preparation and markup*, version 2.0 revised edition, 1988.
- [ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley, 1986.
- [Bar89] David Barron. Why use SGML? *Electronic Publishing, Origination, Dissemination and Design*, 2(1):3–24, April 1989.
- [CMB90] C.M.Sperberg-McQueen and L. Burnard, editors. *Guidelines for the encoding and interchange of machine-readable texts*. Association for Computers and the Humanities, Association for Computational Linguistics, Association for Literary and Linguistic Computing, Text Encoding Initiative P1 draft version 1.1 edition, November 1990.
- [CMS91] D.D. Cowan, E.W. Mackie, and G. de V. Smit. Rita—an editor and user interface for manipulating structured documents. *Electronic Publishing, Origination, Dissemination and Design*, 4(3):125–150, September 1991.

- [CRD87] James H. Coombs, Allen. H. Renear, and Steven J. DeRose. Markup systems and the future of scholarly text processing. *Communications of the ACM*, 30(11):933–947, November 1987.
- [CS86] D.D. Cowan and G. de V. Smit. Combining interactive document editing with batch document formatting. In J.C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 140–153. Cambridge University Press, April 1986.
- [DeR90] Steven J. DeRose. DynaText: Electronic book indexer/browser. *EPSIG News*, 3(4):1–2, December 1990.
- [FSS82] Richard Furuta, Jeffrey Scofield, and Alan Shaw. Document Formatting Systems: Survey, Concepts, and Issues. *Computing Surveys*, 14(3):417–472, September 1982.
- [Gol90] Charles F. Goldfarb. *The SGML Handbook*. Claredon Press, Oxford, 1990.
- [Gro91] Paul Grosso. Arbortext’s SGML-publisher. *EPSIG News*, 4(3):7–8, September 1991.
- [Hay92] Frank Hayes. SGML comes of age. *UnixWorld*, pages 99–100, November 1992.
- [ISO86] ISO International Standards Organization, ISO Central Secretariat, 1 rue de Varembe, CH-1211, Geneva 20, Switzerland. *Standard Generalized Markup Language ISO 8879*, October 1986.
- [KB86] B.W. Kernighan and J.L. Bentley. GRAP – a language for typesetting graphs. *Communications of the ACM*, 29(8):782–792, August 1986.
- [KC75] B.W. Kernighan and L.L. Cherry. A system for typesetting mathematics. *Communications of the ACM*, 18(3):151–157, March 1975.
- [Ker82] B.W. Kernighan. PIC – a language for typesetting graphics. *Software – Practice and Experience*, 12:1–21, 1982.
- [Ker90] Brian W. Kernighan. Issues and tradeoffs in document preparation systems. In R. Furuta, editor, *EP-90, Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography*, pages 1–16. Cambridge University Press, September 1990.

- [Knu84] D. Knuth. *The T<sub>E</sub>XBook*. Addison Wesley, 1984.
- [Lam86] L. Lamport. *L<sub>A</sub>T<sub>E</sub>X, A Document Preparation System*. Addison Wesley, 1986.
- [Miz91] Akira Mizobuchi. Japanese SGML editor and document sharing system. *EPSIG News*, 4(3):3–6, September 1991.
- [Oma87] Paul Oman. Desktop? Yes. Publishing? Not quite. *IEEE Software*, pages 70–77, May 1987.
- [Pia89] G.M. Pianosi. *Waterloo Rita, Document Class Generator*. WATCOM Publications, 415 Philip Street, Waterloo, Ontario, Canada, 1989.
- [PM89] G.M. Pianosi and E.W. Mackie. *Waterloo Rita, Tutorial and Reference*. WATCOM Publications, 415 Philip Street, Waterloo, Ontario, Canada, 1989.
- [Qui89] Vincent Quint. Systems for the manipulation of structured documents. In J. Andre, R. Furuta, and V. Quint, editors, *Structured Documents*, pages 39–74. Cambridge University Press, 1989.
- [RD92] Louis R. Reynolds and Steven J. DeRose. Electronic books. *BYTE*, pages 263–268, June 1992.
- [Smi87] G. de V. Smit. *A Formatter-Independent Structured Document Preparation System*. PhD thesis, University of Waterloo, July 1987. Research report CS-87-40.
- [Smi92] G. de V. Smit. Finding ways to specify semantics for SGML document type definitions. Technical Report CS92-05-00, University of Cape Town, November 1992.
- [Sun90] Sun Microsystems. *SunOS Documentation Tools*, March 1990.
- [USA88] USA Department of Defence, CALS Policy Office, Pentagon, Washington, DC20301. *MIL-M-28001A Markup Requirements and Generic Style Specification for Electronic Printed Output and Exchange of Text*, July 1988.
- [Wri92] Haviland Wright. SGML frees information. *BYTE*, pages 279–286, June 1992.
- [WvV91] Jos Warmer and Hans van Vliet. Processing SGML documents. *Electronic Publishing, Origination, Dissemination and Design*, 4(1):3–26, March 1991.